ターボパスカルVer.6エキスパートマニュアル

ステップアップをめざすプログラマのためのいちばんやさしい解説書

TURBO PASCAL Ver.6

小林侔史

TURBO PASCAL Ver.6

ターボパスカルエキスパートマニュアル

■

小林 侔史

■

目 次

はじめに

TURBO Pascalによるプログラミングはアメリカばかりでなく、日本でも大学のキャンパスを中心に広まっています。その最大の理由は、教育用言語としてPascalが主流になったためです。確かに、構造化手法などとめんどうなことを持ち出さなくても、Pascalで書かれたプログラムは論理構造がとても分かりやすくなります。

しかし、Pascalは教育用で、実用的なソフトウェアの開発はC言語でなければという声もよく聞かれます。どうせソフトウェア開発をC言語でやるのだから、コンピュータ言語の教育もC言語でやるべきだというひとさえいます。

そうでしょうか。

ぼくの今までの経験からいうと、Pascalで上手にプログラムを作れる人なら、C言語でもBASIC、FORTRAN、アセンブリ言語でさえも上手にプログラムが作れるものです。つまり、Pascalでソフトウェア開発ができるものなら、何もC言語を一から勉強する必要はないのです。いつの日かC言語で開発する必要に迫られてから勉強すれば十分です。

せっかく慣れたPascalをソフトウェア開発をしたいがために捨てることはありません。Borland International 社が発売しているTURBO Pascalはパソコンでのソフトウェア開発のためのPascalなのです。しかも、TURBO Pascalには最近の話題であるオブジェクト指向プログラミングも取り入れられ、ますます便利になりました。

この本は本書の姉妹編である「TURBO Pascalトレーニングマニュアル」を読み終えた読者を対象に、実際のパソコンソフトウェア開発の取りかかりのノウハウを解説することを目的としています。

この本によりひとりでも多くのひとがTURBO Pascalによりプログラムを作る喜びを味わってくれれば、これほどうれしいことはありません。

最後になりましたが、TURBO Pascalの発売元である(株)マイクロソフトウェアアソシエイツの企画部長である篠原幸吾さんに大変お世話になりました。また、TURBO Pascalのウィザード(?)である田中真二君には、本書全体に渡りお手伝いいただきました。ここに記して感謝いたします。

1991年7月　ボストンMITキャンパスにて

小林　倖史

まず道具をそろえよう

1 エキスパートへの道

この章では第2章以降に登場するプログラムを理解したり、自分で改造したりするためにそろえておく「道具」について説明します。道具としては当然TURBO Pascalとパソコンが必要ですが、あなたが住んでいる場所によっては参考資料がなかなか手に入らないことがよくあります。それに参考資料では読者が当然知っていると仮定している予備知識がないと、書いてあることがよくわからないこともあるでしょう。特に、あなたがプログラミングの初級者だと、どれも難しいことばかり書いてあり手がかりがなくて困ることもあるでしょう。

しかし、心配は無用です。本書では、各章のサンプルプログラムにはそれぞれ詳しい解説が付けてあります。その解説を理解するための基礎知識は、本書の姉妹編「TURBO Pascalトレーニングマニュアル」(JICC出版局)で十分なはずです。Pascal自体の文法や、TURBO Pascal特有の機能の基礎はこの本に書いてあります。先の章に読み進んでわからなくなったら、いつでも参考にしてください。

ガイドブックあれこれ

1–1■参考資料

(1) TURBO Pascalマニュアル

参考資料としてまず絶対に必要なものはTURBO Pascal ver.(バージョン)6.0のマニュアルです。標準的なPascalの解説書にはPascalの生みの親であるN.Wirth仕様の機能しか書いてありません。TURBO Pascalはパソコンの機能をフルに引き出すため標準的なPascalをかなり拡張しています。実はこの拡張機能こそがTURBO Pascalの価値なのです。このエキスパート・マニュアルでは拡張機能をフルに活用しますから、標準Pascalについて書かれた本では役に立ちません。

TURBO Pascalでは標準ライブラリとしてMS-DOSやグラフィックス関係のものが非常に充実しています。さらに、ver.5.5以降はオブジェクト指向プログラミングへの拡張もなされています。これらすべての機能はマニュアルに詳しく記載されています。

マニュアルはTURBO Pascalを購入すれば添付されていますから、入手の心配はありません。ただ、ここ数年でバージョンアップを繰り返しましたから、ぜひ手持ちの

TURBO Pascal を ver.6.0 にバージョンアップしておいてください。これ以前のバージョンの TURBO Pascal を過去に購入し、ユーザー登録してある人にはすでにバージョンアップの案内がきているはずです。もしも、バージョンアップの案内をそのままにして古いバージョンの TURBO Pascal をまだ持っていたり、案内がこなかったりしたときには以下の住所に連絡をとってみてください。

〒107　東京都港区南青山 7-8-1　小田急南青山ビル
(株) マイクロソフトウェア アソシエイツ　カスタマーサポートセンター
電話：(03)3486-1403
FAX：(03)3486-8905

添付のマニュアルはユーザーズガイド、プログラマーズガイド、ライブラリレファレンス、TURBO Vision ガイドの 4 冊です。このどれもかなりページ数があって読みごたえ (?) があるものです。このマニュアルは TURBO Pascal で何かプログラムを作ろうとするたびに参照するものです。それぞれの内容は次のようになっています。

(a) ユーザーズガイド
インストレーション、統合環境、チュートリアル (自習) ガイド、コマンドラインコンパイラ
(b) プログラマーズガイド
言語定義、ユニットライブラリ、アセンブリ言語、エラーメッセージ集
(c) ライブラリレファレンス
標準手続きをアルファベット順に並べたランタイムライブラリ

(2) MS-DOS マニュアル

TURBO Pascal のマニュアルの次は MS-DOS のマニュアルです。現在発表されている MS-DOS は ver3.3C です。[1]
MS-DOS は特に購入しなくてもワープロソフトなどアプリケーションに付属するもので十分ですが、やはりマニュアルがあったほうがなにかと便利です。MS-DOS を新たに購入すると数万円は確実にかかるので、流用したいところです。このようにバンドル (セットになっている) されているソフトウェアと切り放して TURBO Pascal を走らせるために MS-DOS を流用してもよいのかどうかは議論が分かれるところです。

自分でプログラムを作っている分にはかまいませんが、TURBO Pascal でソフトウェアを開発して、そのソフトウェアに MS-DOS を勝手に付けることは許されません。ただし、MS-DOS を付けなければ TURBO Pascal で開発したソフトウェアは商品として販売することができます。そのための条件は TURBO Pascal に添付されている「ソフトウェアのご使用条件」に詳しく定めてあります。

いずれにしても、MS-DOS を詳しく知りたいのであれば、新たに MS-DOS を購入してマニュアルを手に入れておくことをすすめます。アプリケーションにバンドルしてい

[1]1991 年 6 月末現在、日本電気「PC-9800 シリーズ日本語 MS-DOS(ver3.3C)」。

るMS-DOSを流用するのならば、参考資料として次の2冊がよいでしょう。

「MS-DOSプログラマーズハンドブック」(アスキー)
「標準MS-DOSハンドブック」(アスキー)

MS-DOS自体について知りたいのなら、PUGブックス(JICC出版局)の

「MS-DOS基礎の基礎」
「MS-DOSトレーニングマニュアル」

が分かりやすく書いてあるので、ぜひ一読をおすすめします。

ターゲットにするのは
1–2■使用したパソコン

TURBO Pascalの機能をフルにフルに発揮させるとなると、どうしてもパソコンのハードウェアに依存したプログラムにならざるをえません。そこで問題になるのはどのパソコンを開発ターゲット(対象)にするかです。できればすべての機種のパソコンで動作するプログラムが開発できればそれにこしたことはありません。しかし、現状ではそれは不可能です。現在までに発売されたパソコンの機種は国産のものだけでも100機種以上もあるからです。

ここでは日本電気のPC-9801シリーズのパソコンを対象にしました。

TURBO Pascal自体が日本のパソコンではPC-9801シリーズにしか対応していません。したがって、互換機であるエプソンのPCシリーズのみがそれ以外の使用機種となります。アメリカではIBM-PC互換機＝MS-DOSマシンで、すべて互換の仕様になっていますが、IBM-PC用のTURBO Pascalが用意されています。もしも、あなたが英語を苦にしないのなら、開発ターゲットとしてIBM-PCを使えば、かなりよい環境になるでしょう。今後、IBM社が発表したDOS/Vが本格的に動くようになれば、IBM-PC互換機でも日本語が使用できるようになります。

ひとくちにPC-9801シリーズといっても、CPUの違いで16ビット、32ビットのモデルがありますし、画面表示もノーマルとハイレゾ(ハイレゾリューション、高解像度)モデルがあります。本書のサンプルプログラムは16ビット、ノーマルモデルで動作を確かめてあります。具体的にはPC-9801VMで実行しています。VMモデルはPC-9801シリーズの中でもかなり古いほうに属します。けれども、販売台数がかなりあったため、ソフトハウスが必ず動作を保証するモデルでもあるので、開発ターゲットとしました。モデルの違いで動作しないようなサンプル・プログラムにならないように注意はしましたが、場合によっては正常な動作が保証できないことをお断りしておきます。こういうと、何か無責任のように聞こえるかも知れませんが、すべてのモデルで動作をチェッ

クすることは物理的にも時間的にも無理であることはわかっていただけると思います。また、サンプル・プログラムはできるだけバグ・フリー(バグなし)であるように最大の努力をしたつもりですが、それでもバグが残っている可能性があります。あるサンプル・プログラムが正常に動作しないときに、それがパソコンのモデルの違いに由来するのか、プログラム内のバグに由来するものなのか(ごめんなさい！)を区別するのはとても難しい問題です。

しかし、TURBO Pascalのエキスパートを目指す読者であれば、もし動作しない場合でも必ず解決できるでしょう。その解決の過程もエキスパートになるためのトレーニングと思ってください。なんだ、こんなバグがあるぞとプログラムを訂正できるようになれば、ぼくより実力が十分あるということになります。

PC-9801以外のパソコンを持っている読者はどうしたらよいのでしょうか。世の中のあらかたのパソコン書籍がPC-9801向けの上に、この本までもとがっかりされるかもしれません。しかし、先にも断ったように、パソコンのハードウェアに関係する部分を操作するようなプログラム開発を解説するには、機種を限定せざるをえません。結局、ユーザー数が多いPC-9801シリーズを対象とするという「多数決」でこうなりました。

さて、パソコンのハードウェアに密接に関係するプログラムを作るうえで、ハードウェアの解説書は欠かせません。PC-9801シリーズのパソコンに付属してくる「ユーザーズマニュアル」だけでは不足です。

本当は日本電気からソフトハウス向けに出している「テクニカルマニュアル」があれば一番よいのですが、残念ながら一般ユーザーの手には入りません。そこで、市販の解説書の中からおすすめするとなると、次の2冊がよいでしょう。

「PC-9801解析シリーズ」(秀和トレーディング)
「PC-9801シリーズテクニカルデータブック」(アスキー)

前者は内容的には詳細にわたっているのですが、かなりパソコンのハードウェアについて予備知識のあるユーザーを対象にしているので、「TURBO Pascalトレーニングマニュアル」を終わったばかりの読者には少々難しいかもしれません。そんな読者には後者のほうがよいでしょう。

しかし、これらの解説書はどうしてもなければならないというわけではありません。PC-9801の「ユーザーズマニュアル」にないような情報が必要なときには、本書の中で解説するようにしてあります。

2

DOS ユニット攻略法

MS-DOS コネクション

それではエキスパートへの第一歩として、MS-DOS に関係するプログラムを作ってみましょう。TURBO Pascal はバージョン 4 以上になってから、MS-DOS 関連の手続きや関数は Dos というユニットにまとめられました。

普通、MS-DOS が提供する機能を直接利用するためには、ファンクション・コールに頼らなければならないはずです。そのために本来高級言語である TURBO Pascal の中でも各レジスタを意識して使用せざるをえなかったのです。それが Dos ユニットの充実によって、かなりの機能が標準的な Pascal の手続き・関数として利用可能になりました。

しかし、ここで注意しなければいけないのは、利用可能になったといっても文法的に標準 Pascal に合わせた使用が可能なだけなのだということです。ニコラス・ビルトたちが提案した標準 Pascal には当然、こんなユニットなど用意されていません。Pascal は一種の高級言語の仕様なのですから、OS に直接関連する機能を持たせるわけにはいかないのです。ですから、MS-DOS を利用することを前提として Pascal ができているはずがありません。

つまり、こういうことです。

TURBO Pascal はいかにパソコンでのプログラム作成を容易にするかに特化したしたものなのです。したがって、Pascal が最初に提案された「どのようなコンピュータでも動作する」というコンセプトから離れてしまったともいえるでしょう。BASIC はその生まれたときから個々のパソコンに依存することを運命づけられていたようなもので、それはやむをえないことだったのです。しかし、コンピュータの機種に依存しないようにという Pascal の原点が TURBO Pascal では崩れてしまっていると「文句」をいう人たちも多いようです。

とはいえ、他のコンピュータでもそのまま使えるかどうかの移植性の問題をすべて標準ユニットにしわ寄せして、それを使わなければ標準 Pascal で記述したプログラムが動作するようになっていれば、標準 Pascal に準拠しているといえます。

この章では最初に Dos ユニットにどのような変数や、定数、さらに手続きが用意されているのかを調べて、その後で実際に使用する例題を作ってみることにします。

ファンクション・コールにさようなら?

2-1■Dos ユニット

MS-DOSのファンクション・コールを使う必要があるときには、割り込み手続きIntrかシステム・コール用手続きMsdosを利用します。これらの手続きはパソコン内部の各レジスタを参照しますから、それをTURBO Pascalの変数として定義しておかねばなりません。そのためTURBO Pascalは可変型のレコードをDosユニットの中に定義して持っています。

以下のようなレコード型の変数定義がDosユニットの中ですでに行われているのです。

```
type Registers= record
                  case integer of
                    0: (AX,BX,CX,DX,BP,SI,DI,DS,ES,Flags: word);
                    1: (AL,AH,BL,BH,CL,CH,DL,DH: byte);
                  end;
```

ファンクション・コールを行うプログラム、つまりあなたが作るプログラムの先頭のほうでuses宣言でDosユニットの使用を宣言しておけば、Registers型の変数を定義済みのものとして使うことができます。つまり、

```
program DOS_Example;
....
uses Dos;
....
var Dos_reg : Registers;
....
....
```

とすれば、Dos_regというレコード型変数は内部の各レジスタを参照する変数になります。こうした後で、Msdos手続きなら、

```
Dos_reg.ax := $xxxx;
Msdos(reg);
```

とします。

一方、割り込みintr手続きを使うのならば、

```
Dos_reg.ax := $xxxx;
Intr($21,reg);
```

とします。

つまり、割り込みの$21($マークは16進数を表す)はMS-DOSのファンクション・コールに割り当てられているため、同じことになったのです。

しかし、このようなやり方は、最後の手段とでもいうべきもので、IDE(統合環境)となったver4.0、そして現行のver6.0(1991年6月現在)ではあらかたのMS-DOSのファンクション・コールがDosという名称のユニット、いわばルーチン集に納められているのですから、できるだけそれを利用して標準的なPascalに近い形でプログラムを作るようにしましょう。

さて、次の2つのプログラムを見てください。リスト2-1Aではファンクション・コール、リスト2-1BではDosユニットに入っているGetTime手続きを利用してTimerという名前の手続きでパソコンのカレンダ時計を読み出しています。

■リスト2-1A

```
 1: program LST2_1a;
 2:
 3: uses Crt,Dos;
 4:
 5: var
 6:   startsec,
 7:   endsec   :integer;
 8:
 9: procedure Timer(var second:integer);
10: var
11:   reg : Registers;
12:
13: begin
14:   reg.ax := $2c00;
15:   Msdos(reg);
16:   second := Trunc(reg.cx/256)*3600+(reg.cx mod 256)*60
17:     +Trunc(reg.dx/256);
18: end;{of Timer}
19:
20: begin
21:   CLrScr;
22:   Timer(startsec);
23:         DeLay(1000);
24:   Timer(endsec);
25:   WriteLn( 'Time: ',endsec-startsec);
26: end.{of Main}
```

■リスト2-1B

```
 1: program LST2_1b;
 2:
 3: uses Crt,Dos;
 4:
 5: var
```

```
 6:        starthour,startmin,startsec,startcent : word;
 7:        endhour,endmin,endsec,endcent         : word;
 8:        elapsedsec : integer;
 9: begin
10:    CLrScr;
11:        GetTime(starthour,startmin,startsec,startcent);
12:        DeLay(1000);
13:        GetTime(endhour,endmin,endsec,endcent);
14:        elapsedsec := 3600*(endhour-starthour)
15:         +60*(endmin-startmin)
16:        +(endsec-startsec);
17:        WriteLn( 'Time: ',elapsedsec);
18: end.
```

まず、リスト2-1Aでは14行目でAレジスタに$2c00を設定して、ファンクション・コールを実行するとMS-DOSでは各レジスタの内容は次のようになります。

	CH	CL
Cレジスタ	時(0～23)	分(0～59)
	DH	DL
Dレジスタ	秒(0～59)	1/100秒
	上位8ビット	下位8ビット

これを利用して時間を測定するのですが、ほとんど同じようにしてリスト2-1Bのプログラムでは11、13行目で簡単に時間を読み出しています。ただし、リスト2-1Bのほうでは Aレジスタにファンクションコールの番号である$2c00の設定は行っていません。それはDosユニット中のGetTimeの仕事だからです。

MS-DOSの仕様とは異なり、PC-9801シリーズでは1/100秒はカレンダ時計でサポートされていません。どうしても、1秒以下の単位で測定をしたいときには第3章の割り込みを利用して、インタバル・タイマを使わなければなりません。

1/100秒が計れなくても、ある計算の所用時間を測定したければ、その計算を100回繰り返して測定し、それを100分の1にすれば実用上問題ないでしょう。ここでは変数startsecに開始時刻、endsecに終了時刻を秒単位で読み取って最後に引き算してforループを1000回繰り返したときの所用時間を測定しています。

これはまったく簡単な例題ですが、このファンクション・コールを用いると自分でプログラムを作るには少々面倒なルーチンでも、MS-DOSを利用していろいろ作れそうです。MS-DOSのマニュアルを参考にして、かたっぱしからファンクション・コールを試してみてください。

ただし、ディスク入出力関係のファンクション・コールを試すときには、オリジナル・ディスクのバックアップを作ってからトライしないと、大変なことになりかねないので注意が必要です。それさえ気をつければ、ファンクション・コールでパソコンは壊れたりしませんから、自由に操作してかまいません。

とっておきのノウハウ

2–2■コマンドパラメータの引き渡し

次にTURBO Pascalで作ったプログラムにMS-DOS上からパラメータを引き渡すテクニックを練習してみましょう。むろん、プログラムの中にReadLnを入れて、データを読み込むようにしてもよいのですが、MS-DOSのコマンドらしいプログラムを作るためにはこのコマンド・パラメータの引き渡しが必要になります。

ここではuses宣言なしで使用可能なStandardユニットに入っているコマンドパラメータの引き渡しを利用します。本来Standardユニットには、主に標準Pascalの手続きや関数が入っているのですが、ここで使用する手続きはどちらかというとDosユニットにあるべきものなので、あえて取り上げてみました。

本題に入る前に(すでにわかっていることとは思いますが)、TURBO Pascalでプログラム開発をして実際に走らせるまでの手順を考えてみましょう。

プログラム開発のやりかたはひとによって違いますが、細かいところはとにかく大きな流れとしてはだれでも次のようにやっているはずです。

(1)TURBO Pascalを起動する
(2)Editウィンドウでプログラムを作成する
(3) コンパイルして実行する

この(3)の手順のところで、パソコン上で実行、つまり走らせるときの方法は2通りあります。コンパイル結果をメモリ・モデルとするかEXEファイル・モデルとするかです。普通なにも指定していなければ、コンパイル結果はメモリ・モデルとなります。メモリ・モデルでは、他のプログラムをコンパイルしたり、TURBO Pascalを終了するとそのプログラムのコンパイル結果は失われてしまいます。したがって、メモリ・モデルはTURBO Pascal内のみだけで実行可能で、MS-DOSからは実行させることはできません。これでは、便利なプログラムを作っても、TURBO Pascalをいちいち起動しなければ使えないことになり、とても不便で困ります。

そこで、TURBO PascalでコンパイルするときにCompile(コンパイル)メニューを選択し、Destinationを選択してリターンキーを押すと、MemoryとDiskに切り替わります。ここでDiskとすれば今作ったプログラムと名前が同じで、拡張子がEXEのEXEファイル・モデルを作ります。こうすると、たとえばSAMPLE.PASというプロ

グラムのコンパイル結果として、SAMPLE.EXEというコマンド・ファイルがディスク上に残ります。

EXEファイル・モデルはMS-DOSから直接実行可能です。実はそれを(MS-DOSから直接実行できるファイルを)コマンドと呼んでいるのです。このMS-DOSのコマンドとしてはdirとかcopyなどがおなじみのものですね。

MD-DOSのコマンドには内部コマンドと外部コマンドの2種類があります。内部コマンドとは早い話がdirで画面表示されないコマンドです。一方、外部コマンドとはformatのようにdirで名称が見ることのできるコマンドです。TURBO Pascalのプログラムをコンパイルしてディスク上に作った実行可能モジュールはこの外部コマンドとなります。したがって、SAMPLE.EXEは外部コマンドということになります。

さて、コマンドはMS-DOSモードでのみ実行できますから、ディスク上に作られたSAMPLE.EXEを走らせるにはTURBO PascalのIDEのFileメニューのOS shellでMS-DOSのモードにするか、TURBO Pascalをいったん終了してMS-DOSに戻らないと実行できません。上の例ならば、OS shellを選択したか、TURBO Pascalを終了させた状態でディスプレイにA>が表示され、なおかつSAMPLE.EXEがAドライブ上に存在しているのなら、

```
A>SAMPLE
```

と入力してリターンキーを押せば確かにMS-DOS上でSAMPLEが実行されます。

ここまではすでにTURBO Pascalでプログラム経験を重ねたひとにはおなじみの話です。実はコマンド・パラメータの話はここからです。ただMS-DOS上でプログラムが動いたと喜ぶひとはいません。それをもうひとひねりしてみようというわけです。

いま、このSAMPLEというプログラムがなにかファイル名を読み込んで、そのファイルになんらかの処理を加える機能を持っているとします。そのファイルとしてFILE1〜10まで10本あるとしたらどうしますか。

普通の作り方でSAMPLEというプログラムを作ると、プログラムのどこかで処理すべきファイルの名前を入力させるようにするでしょう。こうして作られたEXEファイルを走らせるには、A>SAMPLEとして実行し、処理すべきファイル名の入力が要求されたところでFILE1と入力します。処理が終ると再びA>が表示されますから、またSAMPLEと入力し、しばらくしてから今度はFILE2と入力します。しかし、こんな調子で10回同じことをやらされるのではかないません。それでも忍耐力のある人ならなんとかできるでしょうが、20本、30本となったらどうしますか。

考えられる手は処理すべきファイル名のすべてをプログラムの最初のほうであらかじめ入力させることでしょう。そして、たとえばファイル名に使いそうもない記号、たとえば@とか#を最後に置いてやれば不特定の本数のファイル名を入力させることができ

ます。しかし、もしそのプログラムをMS-DOSの外部コマンドとして使用するのだとすると、これでは外部コマンドというよりは、いわゆるアプリケーション・ソフトウェアのようになってしまいます。

そこで、MS-DOSのバッチファイルを使ってなんとか自動的にFILE12、3、...の処理をやらせたいところですが、毎回処理すべきファイル名が違うのではお手上げです。バッチの手順としてキーボード操作をそのままに、

```
SAMPLE
FILE1
SAMPLE
FILE2
  ⋮
```

とできればよいのですが、そうはいきません。SAMPLEというEXEファイルが読み込むべきパラメータをバッチファイルに記入しておいても、バッチファイルはMS-DOSのコマンドを並べるべきものですから、キー入力の手順を並べてもだめなのは当然です。

しかし、処理すべきファイル名をSAMPLEに対する入力ではなく、MS-DOSのコマンドに対するコマンド・パラメータとして引き渡せれば、次のようなバッチファイルを作り実行させることができそうです。

```
SAMPLE   FILE1
SAMPLE   FILE2
SAMPLE   FILE3
  ⋮
```

これができればパソコンに汗をかかせて(?)、あなたは涼しい顔でゆっくりコーヒーを飲みながら処理の終了を待つことができます。

余談ですが、バッチファイル程度の簡単なファイルを作るにはなにもエディタを使ったり、ましてエディタの化石とまでいわれているMS-DOS付属のEDLINを起動するまでもありません。もし、このバッチファイルのファイル名がSHORI.BATであれば、

```
A>COPY CON SHORI.BAT
```

としてから必要な内容を入力し、最後に^Z[1]を入力すれば簡単に作れます。この方法はCONFIG.SYSを書き換えるときにも便利ですから覚えておくとよいでしょう。

さて、MS-DOSにはコマンド・パラメータ用に最高128文字の長さのバッファ領域が確保されています。その場所は絶対番地のcseg：$0080です。csegはコードセグメン

[1]コントロール・キーとZを同時に押すことをこう表記することがあります。Ctrl-Zとも表わされます。

ト、$0080 は 80 番地を表します。しかし、TURBO Pascal でコマンド・パラメータをを引き渡すプログラムを作るためにはそんなことは知らなくてもかまいません。ちょっとパソコンに詳しいひとがコードセグメントのどこそこにこれこれの情報が入っているなどといっているのを聞くと、すごいなあなどと思っているのではありませんか。こんな知識は TURBO Pascal のマニュアルにいくらでも書いてあるようなことですから、感心することはありません。知識をひけらかすよりも、どうプログラムを作るかが一番大切なのです。

さて、リスト 2-2 を見てください。とても短いプログラムですが、これでパラメータ引き渡しはどうすればよいのかよくわかるでしょう。8、9 行目に現れる ParamCount はコマンドラインでプログラムに引き渡されたパラメータの個数、ParamStr は指定された個数目のパラメータを返す標準関数です。

■リスト 2-2

```
 1: program LIST2_2;
 2:
 3: var
 4:   count : word;
 5:
 6: begin
 7:   WriteLn('Start Listing');
 8:   for count := 0 to ParamCount do
 9:   WriteLn('No',count:1,' ',ParamStr(count));
10:   WriteLn('End Listing');
11: end.
```

```
A>b:list2-2 1234567890 ABCDEfghi S.I.T
Start Listing
No.0 b:¥LIST2-2.EXE
No.1 1234567890
No.2 ABCDEfghi
No.3 S.I.T
End Listing
```

■図 2-1(実行例)

プログラムはとても簡単ですからリスト内の書き込みを見れば、どうなっているかすぐに理解できるでしょう。図 2-1 はこのプログラムを走らせたときの結果です。コマンド・パラメータとして 1234567890、ABCDEfghij と S.I.T を渡してみました。ここで注目すべきことは大文字と小文字が区別されていることです。普通、MS-DOS を使用

しているときには大文字も小文字も区別なく使えるので、つい忘れがちですから気をつけましょう。コマンド・パラメータを利用してプログラムに文字そのもの、つまり文字列を引き渡すときにはこのことを注意しておく必要があります。さもないと、原因不明のバグが出たりします。

その対策としてはプログラム内部でコマンド・パラメータをUpcase関数を使って、すべて大文字に直してから受け取れるようにしておくのがノウハウです。いずれにしても、このコマンド・パラメータのテクニックは知っていると簡単に便利なプログラムが作れることが多いので、たいへん利用価値があります。TURBO Pascalのバージョン6以上では標準関数がとても充実していますので、今までMS-DOSのファンクションコールを使ったり、あるいはMS-DOS特有のパラメータの番地を参照したりしないと実現できなかった機能をあなたのプログラムに実現することができるようになりました。「こんな機能はどうやれば実現できるのかな」と思ったら、TURBO Pascalマニュアルの Dosユニット関係の関数や手続きを調べてみてください。

Save the Earth
2–3■プログラム内からのリダイレクト

プリンタに大量の印刷出力を行うプログラムを開発していると、テストのたびに裏を「メモ用紙」にしか使えない失敗プリントが大量に吐き出されます。これではまったく森林資源のムダ使いで、最近問題になっている環境破壊にぼくらが手を貸してしまいかねません。なにしろ、最近の紙の消費量の爆発的増加の原因はコンピュータ出力用紙がその原因のナンバーワンなのですから。

それに、プリンタでしょっちゅう出力していたのでは時間がかかってたまりません。そこで、必要のないときにはスクリーンに表示するだけで、印刷したいときだけプリンタに出力できるようにすればいいでしょう。このように、出力する装置をディスプレイやプリンタ、さらにディスクへと変更することをリダイレクトと呼びます。

MS-DOSにはこのリダイレクト機能が備わっていますので、TURBO PascalでEXEファイルを作り、MS-DOSの管理下で走らせるのなら、今までプリンタに出していた出力をディスプレイに表示させることができます。たとえば、

```
A>SAMPLE > CON
```

とすれば出力はコンソール、つまりディスプレイに表示できます。

一見、これで不都合は何もないようですが、デバッグ段階にあるプログラムだとどうでしょうか。EXEファイルを走らせるためにはいったんTURBO Pascalを終了させなければなりません。TURBO Pascalを終了してからMS-DOSに戻り、EXEファイル

を走らせてからまたTURBO Pascalを起動してはプログラムを書き換えてEXEファイルを作り、また同じことをするというのではやってられません。

もうひとつの考え方は、WriteやWriteLnのパラメータにデバッグが済むまでCONを使い、終ったところでTURBO Pascalのエディタの検索機能を使ってCONをPRNに置き換える方法です。これも悪くはありませんが、検索して置換するときに、置き換えてよいものと悪いものをいちいち判断しなければなりません。長時間デバッグを続けて「頭が熱く」なってくるとこの判断も誤りがちになります。

ここで提案する方法はTURBO Pascalに用意されているファイル変数に対するAssign手続きを利用するものです。Assignの標準形式は、次のとおりです。

```
Assign(fil,str)
```

strという文字で表されるファイル名をfilに割り当てるのが、この手続きの働きです。

■リスト2-3

```
 1: program LIST2_3;
 2: uses    crt;
 3:
 4: var
 5:   shuturyoku : Text;
 6:   souchi     : Char;
 7:   Line       : String[40];
 8:
 9: begin
10:   CLrScr;
11:   Write(' 出力は（S）スクリーンですか（P）プリンタですか？');
12:
13:   repeat
14:     readLn(souchi);
15:     souchi := Upcase(souchi);
16:   untiL souchi in ['S','P'];              {  s,p 以外は再入力  }
17:
18:   WriteLn;
19:
20:   if souchi='S'
21:     then Assign(shuturyoku,'CON')         {  コンソールへ  }
22:   eLse Assign(shuturyoku,'PRN');          {  プリンターへ  }
23:
24:   Rewrite(shuturyoku);                    {  出力モードでオープン  }
25:
26:   ReadLn(Line);                           {  出力する文字を読み込む  }
27:
28:   WriteLn(shuturyoku,Line)                {  アサインされた出力先へ  }
29: end.
```

リスト2-3を見てください。21行目と22行目でshuturyokuというファイルにCONまたはPRNというファイル名を指定しています。つまり、21行目ではshuturyokuというファイルはCON(コンソール)になりますし、22行目ではPRN(プリンタ)になります。このプログラムでは14行目で入力された文字がSならスクリーン(Screen)に、Pならプリンタ(Printer)に26行目で入力された文字列を出力します。

■リスト2-4

```
 1: program LIST2_4;
 2:
 3: Uses
 4:   Crt;
 5:
 6: var
 7:   shuturyoku_file,
 8:   shuturyoku_souchi : text;
 9:   TextLine          : string[255];
10:   fiLename          : string[14];
11:   souchi            : char;
12:
13: begin
14:   CLrscr;
15:   Write('出力装置は（S）スクリーンですか、',
16:         '（P）プリンタですか？');
17:   repeat
18:     ReadLn(souchi);
19:     souchi := upcase(souchi);
20:   untiL souchi in ['S','P'];           { s,p以外は再入力 }
21:
22:   WriteLn;
23:
24:   if souchi='S'
25:     then Assign(shuturyoku_souchi,'CON')   { コンソールへ }
26:     eLse Assign(shuturyoku_souchi,'PRN');  { プリンタへ   }
27:
28:   Rewrite(shuturyoku_souchi);              { 出力モードでオープン }
29:
30:   Write('ファイル名を入力してください。'); { 出力するファイル名を入力 }
31:   ReadLn(fiLename);
32:
33:   Assign(shuturyoku_file,fiLename);        { 出力するファイルをオープン }
34:   Reset(shuturyoku_file);
35:
36:   whiLe not Eof(shuturyoku_file) do
37:   begin
38:     ReadLn(shuturyoku_file,TextLine);      { ファイルからTextLineに読む }
39:     WriteLn(shuturyoku_souchi,TextLine);   { TextLineから出力装置へ }
40:   end;{of whiLe}
41: end.{of program}
```

こんな例では実用にほど遠いので、このテクニックを応用してテキストファイルのリストをスクリーンかプリンタのどちらか好きなほうに出力させるプログラムを作ってみましょう。

リスト 2-4 を見てください。29 行目まではリスト 2-3 とほとんど同じです。31 行目で出力するファイル名を指定し、ファイルが終りになるまで 1 行ずつ次々にスクリーンまたはプリンタに出力します。ファイル名は拡張子を含めて 14 文字以内にしています。これは 10 行目での指定でそうなっているだけですから、あなたの都合で簡単に変更できます。

それから考えておかなければならないのは、ファイルの内容が 1 画面分以上あるときにスクリーンへ表示すると、次々に内容が流れるように表示されてしまって内容が読めないときの対策です。それには適当な行数スクリーンに表示したところでなにかキー入力があるまで次の表示を待機状態にさせればよいでしょう。そこで 35 行目に、

```
j:= 1;
```

37 行目と 38 行目の間に、

```
if devices = 's' then
  begin
    if j>22 then
      begin
        repeat until KeyPressed;
        j := 0;
      end
    else
      j := j + 1;
```

を挿入するとスクリーン表示は何かキーを押下するごとに、次々と 1 画面づつ表示するようになります。むろん、j を integer で var 宣言するのを忘れないでください。

ファンクション・コールのフル活用

2-4■ディレクトリ表示

ファイル処理用のアプリケーション・プログラムの中でファイルの一覧表、つまりディレクトリを表示したいことがよくあります。MS-DOS の中からは dir と入力することに相当しますが、TURBO Pascal では Dos ユニットに用意されている手続き FindFirst と FindNext を組み合わせることとなります。

ここではTURBO Pascalのファンクション・コールの使用例として、あえてこれらの手続きを使用せずに作った例を紹介します。FindFirstとFindNextによりあっさり同じ機能がどう実現できるのかも示しましょう。

それではまずファンクション・コールからです。ファンクション・コールについてはいろいろ参考書がありますが、やはりどうしてもMS-DOSのマニュアルをかたわらにおいて調べながらでないと無理でしょう。それにしても、やれデータセグメントだのオフセットだのと、パソコンの内部的なことを知らないと苦労させられます。けれども、ここで解説する最初の例題がよくわからないからと絶望する必要はありません。解説をざっと読んで「そんなものかな」という程度の理解で十分です。いずれプログラムを作って、あれこれ実験しているうちにわかるようになります。

前置きはこのぐらいにして、本題のファンクションコールの例題をやってみましょう。リスト2-5を見てください。

■リスト2-5

```
 1: program LIST2_5;
 2:
 3: Uses
 4:         Crt,
 5:         Dos;
 6:
 7: type
 8:   Str12      = string[12];
 9:
10: var
11:    fiLe_shitei,
12:    fiLe_mituke : Str12;
13:    mituke      : booLean;
14:
15: {*****************************************************}
16:
17: Procedure DTA_Address(VAR DTA_Segment,
18: DTA_Offset : integer);
19:
20: var
21:    dos_reg   :  Registers;
22:
23: begin
24:   with dos_reg do
25:     begin
26:       ax := $2F shL 8;
27:      Msdos(dos_reg);
28:        DTA_Segment := es;
29:        DTA_Offset := bx;
30:     end;
31: end;
```

```
32:
33: {***************************************************}
34:
35: Function DTA_FiLename  :  Str12;
36:
37: var
38:    DTA_Segment,
39:    DTA_Offset,
40:    i              :  integer;
41:    DTA_char       :  char;
42:    fiLename       :  Str12;
43:
44: begin
45:   DTA_Address(DTA_Segment,DTA_Offset);
46:   fiLename := '';
47:   i         :=30;
48:   DTA_char      :=chr(mem[DTA_Segment:DTA_Offset+i]);
49:   whiLe DTA_char <> chr(0) do
50:    begin
51:      fiLename := Concat(fiLename,DTA_char);
52:      i  :=  i+1;
53:      DTA_char :=  Chr(mem[DTA_Segment:DTA_Offset+i]);
54:   end;
55:   DTA_FiLename := fiLename;
56: end;
57:
58: {***************************************************}
59:
60: Procedure First_FiLe(      path_addr : Str12;
61:                        var  fiLename : Str12;
62:                        var  mituke   : booLean);
63:
64: var
65:   dos_reg     :     Registers;
66:   fLg         :     byte;
67:
68: begin
69:   path_addr  :=  Concat(path_addr,chr(0));
70:   with dos_reg do
71:   begin
72:     ax  :=  $4E shL 8;
73:     ds  :=  (Seg(path_addr));
74:     dx  :=  (Ofs(path_addr)+1);
75:   end;
76:   Msdos(Dos.Registers(dos_reg));
77:   fiLename  :=  '';
78:   fLg        :=  dos_reg.fLags AND 1;
79:   if fLg  =  0 then begin
80:     mituke  :=  TRUE;
```

```
 81:     fiLename   :=   DTA_FiLename;
 82:    end
 83:      eLse mituke   :=   FALSE;
 84:    end;
 85:
 86: {****************************************************}
 87:
 88: Procedure Dir_Next(      path_addr : Str12;
 89:                     var  fiLename : Str12;
 90:                     var  mituke   : booLean);
 91:
 92: var
 93:    dos_reg   :    Registers;
 94:    fLg       :    byte;
 95:
 96: begin
 97:    path_addr   := Concat(path_addr,chr(0));
 98:    with dos_reg do
 99:    begin
100:      ax  :=  $4F shL 8;
101:      ds  :=  (Seg(path_addr));
102:      dx  :=  (Ofs(path_addr)+1);
103:    end;
104:    Msdos(Dos.Registers(dos_reg));
105:    fiLename   :=  '';
106:    fLg       :=  dos_reg.fLags AND 1;
107:    if fLg = 0 then begin
108:      mituke   :=  TRUE;
109:      fiLename :=  DTA_FiLename;
110:    end
111:      eLse mituke   :=   FALSE;
112: end;
113:
114: {****************************************************}
115:
116: begin
117:    CLrScr;
118:    fiLe_shitei   :=   '*.*';
119:    First_FiLe(fiLe_shitei,fiLe_mituke,mituke);
120:    if mituke then begin
121:      Write(fiLe_mituke:20);
122:      repeat
123:        Dir_Next(fiLe_shitei,fiLe_mituke,mituke);
124:        if mituke then Write(fiLe_mituke:20);
125:      untiL NOT mituke;
126:    end;
127: end. {  Main  }
```

このリストは指定ファイルのディレクトリを表示させます。5行目でDosユニットの使用を定義していますから、すでに2-1節で説明したようにファンクション・コールを使用する前のレジスタはこの中で定義済みになっています。プログラム中で使用されているdos_regというレジスタを表す変数の型であるRegistersになっています。

ここで使うファンクション・コールは$2F、4E、4F の3つです。まず$2F が手続きDTD_Addressで使用されています。この手続きはDTA_SegmentとDTA_Offsetに DTAのセグメントとオフセットを返すものです。DTA(Data Transfer Area、データ・トランスファ・エリア、データ転送アドレス)はファイルと入出力するときのレコードをしまっておく場所なのです。MS-DOSは直接ファイルに対して入出力するのではなく、このDTAに対して入出力しますから、ディレクトリを取ったときにもまずここにディレクトリが入れられます。それを後で取り出すためにまずDTAのアドレスが必要になります。

まず、21行目で例のレジスタに定義した変数としてdos_regを宣言しています。26行目はax：=$2F00としても同じなのですが、$2Fつまり$002Fにしておいてから、左に8ビットシフトしているのです。shlとshrは結構応用範囲が広い演算ですから覚えておくとよいでしょう。27行目のファンクション・コールによりes(エクストラ・セグメント)にはDTAのセグメント・アドレス、bx(Bレジスタ)には同じくオフセット・アドレスが格納されます。

次に関数DTA_Filenameは8行目で定義した12文字長の文字としてDTAからファイル名を手に入れてきます。46行目は、ファイル名を入れる変数filenameをクリアするために必要です。47行目はファイル名がDTAの30個目のメモリに入っていることを示します。DTAの構造はMS-DOSのマニュアルから探しました。49行目から54行目でファイル名を1文字ずつ拾ってはfilenameにConcat手続きを使ってつないでいます。こうして55行目でファイル名が関数として得られます。

60行目の手続きFirst_Fileでは引数path_addrに一致するファイル名filenameを返します。論理変数mitukeは一致するファイル名があればTRUEとなり、なければFALSEになります。

69行目でpath_addrにchr(0)を結合しているのは文字列としてのPath_addrをASCIZ形式にするためです。MS-DOSではASCII文字列に00Hを付け加えたものをASCIZ文字列と呼びます。MS-DOSは文字列の長さをどこかに持っているようなことはしません。その代わりに、00Hが付けられていると、そこでその文字列が終っていると認識するのです。ですから、MS-DOSはある文字列の長さを知るためには00Hに当たるまで文字を次々に読み出さなければなりません。なお、TURBO Pascalでは文字列の先頭の1バイトにその文字列の長さが入っています。

さて、70～76行までのファンクション・コールで何が手に入るのでしょうか。このファンクション・コールは一致するファイルを検索します。まず、AHつまりAX(Aレ

ジスタ)の上位8バイトに$4Eをセットします。DSとDXにはこれから検索しようとするファイル名の入っているアドレスのセグメントとオフセットをそれぞれセットします。78行目はエラーなしでファンクション・コールから戻ってきたかどうかをバイト型の変数flgに入れています。flagの第1ビット目はキャリー・フラッグで、これがセットされているとエラーがある、つまり検索しようとしたファイル名がないということになります。

1とANDをとっていますから、これが0になればセットされていないのでPath_addrに一致するファイルがあったのです。flagのキャリーがセットされていれば、そのファイルはディレクトリにはなかったことになります。見つかったかどうかを論理変数mitukeに格納しておきます。見つかったときには81行目で、先ほど作っておいた関数、DTA_Filenameを使ってそのファイル名を手に入れます。

このFirst_Fileでは一致するファイル名の先頭の1個しか取り出せません。たとえば、MS-DOSでdir *.*としたときでも1個だけしか出てこないことになります。これでは困るので外にも一致するファイル名がないかどうかを調べるための手続きが88行目から始まるDir_Nextです。

引数や97行目のASCIZ文字列に直す操作は前のFirst_Fileと同じです。今度はファンクション・コールの$4Fを利用します。このファンクション・コールでは次に一致するファイル名を検索します。107行目でキャリーがセットされているかどうかを調べています。キャリーがセットされていなければmitukeはTRUEとなりまだ一致するファイル名があり、セットされていればmitukeはFALSEになりもうこれ以上一致するファイルがないことを示します。

これでやっと準備が整いました。116行目からがメインプログラムです。ちゃんと動作するかどうか調べるために、118行目でワイルドカード指定を使ってすべてのファイルを表示させるようにしました。つまり、MS-DOSでdir *.*を入力したときと同じにしてみます。First_Fileを呼んでmitukeがTRUEであれば少なくともひとつはファイルがありますから、まずそれを表示します。そして122～126行目で繰り返し、ファイルがある限り表示を続けます。

図2-2はこのプログラムを動かしてみたときの結果例です。むろん、この例の表示はあなたのファイルに何が入っているかによってまったく変わってくることはいうまでもありません。また、この結果を見てもっときれいに表示したらどうかと思うでしょう。ここではどうやってMS-DOSのファンクション・コールを使うのか、またMS-DOSの内部データはどうなっているのかとか、レジスタ関係はどうアクセスするのかなどを示すまでにしてあります。表示についてはあなたの好みに作り替えてみてもおもしろいでしょう。

```
    PUG.LOG                TEST.PAS              READ.ME        LIST2-1A.EXE
 LIST2-1B.EXE           LIST2-2.EXE          LIST2-3.EXE        LIST2-4.EXE
  LIST2-5.EXE           LIST2-6.EXE          RSDEMO.EXE         KOBLIST.EXE
       LIST
```

■図 2-2(実行例)

レジスタをアクセスしてファンクション・コールを使用するだけでなく、特定のパソコンの機種に依存するBIOSまで使用することも可能です。しかし、そこまでやってしまうとプログラムの移植性がかなり低下してしまうので、高級言語を使用した意味がなくなってしまいます。パソコンの内部的な機能を利用するとしても、せいぜいファンクション・コールまでにとどめるのが正しいプログラマーではないでしょうか。

さて、今度は同じことをDosユニットに用意されている手続き、FindFirstとFindNextを使用して実現してみましょう。まず、手続きFindFirstはどんなものか見ておきましょう。標準の呼び出し形式は、

```
FindFirst(パス:string;属性:word;var  s:SearchRec);
```

です。ここでパスは*.*や*.pasのようなディレクトリマスクです。属性はファイルの属性を表すパラメータで、Dosユニット内で定義済みの次のような定数です。それぞれの属性の意味を定義の後ろに簡単に書いておきました。

```
const
  ReadOnly = $01;{読み出し専用}
  Hidden   = $02;{隠しファイル}
  SysFile  = $04;{システムファイル}
  VolumeID = $08;{ボリューム名}
  Directry = $10;{ディレクトリ}
  Archive  = $20;{通常のファイル}
  AnyFile  = $3F;{上記すべてのファイル}
```

この属性は定義済みなのですから、usesによりDosユニットの使用を宣言した後にはこのまま使うことができます。また、ファイルの属性は、

```
SysFile + Archive
```

のように、組み合わせて使用することができます。

3番目の引数sもDosユニット内で定義されているレコード型、SearchRecのタイプの変数です。定義の形式とその簡単な内容を次に示しておきます。

```
type
  SearchRec = record
               Fill:array[1..21] of byte;{MS-DOS専用データで使用禁止}
               attr:byte;{ファイル属性}
               Time:longint;{圧縮されたタイムスタンプ}
               Size:longint;{バイトサイズ}
               Name:string[12];{ファイル名}
            end;
```

Time は4バイトに圧縮された日付と時刻ですから、普通の日付と時刻に変更するにはやはり Dos ユニットに用意されている UnpackTime 手続きを利用します。呼び出しの形式は、

```
UnpackTime(Time:longint; var DT:DateTime);
```

です。ここで Time は当然 SearchRec タイプの Time 項目ですが、DateTime は次のような定義済みレコード型です。各項目の内容は説明するまでもないでしょう。

```
DateTime = record
              Year,Month,Day,Hour,Min,Sec:word
           end;
```

以上の定義済みの定数と変数を使用して、たとえばシステムファイルと保存ファイルのすべてを指定して探すのなら、次のようにすればよいのです。

```
FindFirst('*.*',SysFile+Archive,SearchedData);
```

しかし、この FindFirst だけでは指定に該当する最初のファイルが得られるだけです。次々と該当ファイルを探すためには手続き FindNext が必要です。手続きの定義は、

```
FindNext(var s:SearchRec)
```

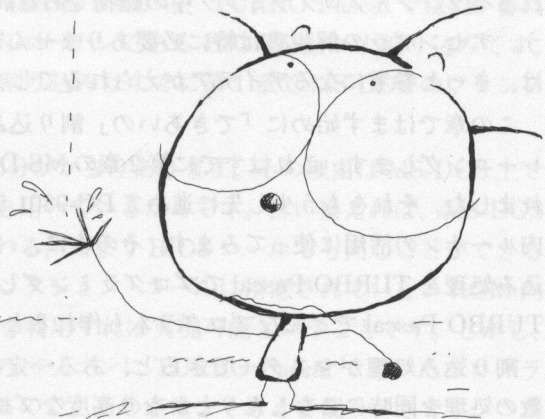
です。引数の s は FindFirst の s と同じ名前でなければなりません。したがって、次のように呼び出せば、FindFirst で探されたファイルの次の該当ファイル名がレコード型変数の項目、Name に返されます。

```
FindNext(SearchedData.Name);
```

それでは該当ファイルが存在しないときにはどうなるのでしょうか。その場合は Dos ユニットで定義されている整数型の変数 DosError に場合に応じて数値が返されるのでそれを監視しておけばよいのです。それが0以外の値になったときには何らかのエラーが生じたことがわかります。DosError に値が返されるのはファイル関係に限りませんが、たとえばファイルが存在しなければ2、パスが存在しなければ3、そしてもう該当ファイルがなければ18になります。18になるときには FindNext で次のファイルが存

在しないときにあたります。ここまでの予備知識があれば、先ほどのリスト 2-5 と同じ機能を実現するプログラムが簡単に作れます。リスト 2-6 を見てください。

■リスト 2-6

```
 1: program LIST2_6;
 2:
 3: Uses
 4:         Crt,
 5:         Dos;
 6:
 7: type
 8:    Str12        = string[12];
 9:
10: var
11:         fiLe_shitei : Str12;
12:         dirinfo : SearchRec;
13:
14:
15: begin
16:    CLrScr;
17:    fiLe_shitei  :=    '*.*';
18:         FindFirst(fiLe_shitei,AnyFile,dirinfo);
19:         while DosError = 0 do
20:         begin
21:         Write(dirinfo.name:20);
22:         FindNext(dirinfo);
23:         end;
24: end. {  Main  }
```

3 迷惑かまわず割り込もう

市販のパソコンソフトを見ていると、どうしたらこんなプログラムが書けるのだろうと不思議な気がしませんか。特にひとつの処理をやりながら、ある特定の条件が起こるといままでとは違う処理に移り、それが終るとまたもとの処理に復帰する割り込み処理などは一体どうやってプログラムするのか実に不思議ですね。こんな複雑なプログラムを作るには、きっとアセンブラとかいうものの知識が相当ないと駄目なのではないかとあきらめてはいませんか。また、あるいは最近人気が出てきたC言語を使わないとできないなどというひとまでいます。C言語はアセンブラとの相性がよいので、ソフト・ハウスのプログラマーに愛用されています。そこでむずかしいプログラムはC言語でないと作れないなどという困った(誤った)ことを信じているひとがいるのです。

そのせいか、TURBO Pascalは「素人向け」あるいは「教育専門」、C言語は「プロ向け」という誤った認識が生まれてきています。しかし、TURBO Pascalがそんな奥の浅いものであったなら、あれほど優れたソフトがひしめいているアメリカのパソコン・ソフト市場でベストセラーになるはずがありません。Borland社のTURBO Pascal設計者はその点についてもしっかり配慮してくれています。

それではTURBO Pascalを使って、「これができればプロ・プログラマー」といわれるパソコン・プログラミング中の難物である割り込み処理にチャレンジしてみましょう。アセンブラの解説書は特に必要ありませんが、深いところまで理解したい読者には、きっと参考になるアイデアがえられるでしょう。

この章ではまず始めに「できあいの」割り込み処理であるIntr手続きの使用法をトレーニングします。これはすでに第2章のMS-DOSコネクションのところでちょっとふれました。それをもう少し先に進めてPC-9801シリーズのパソコンの持っているROM内ルーチンの活用に使ってみます。その次に、ベクタテーブルを利用した本格的な割り込み処理をTURBO Pascalでプログラミングします。これが理解できるようになるとTURBO Pascalでどんなプログラムも作れるという自信が出てくると思います。

割り込み処理がマスターできると、ある一定の時間間隔でデータを計測したり、複数の処理を同時にこなしたりとかなり高度なプログラムが書けるようになります。やっとTURBO PascalのMS-DOS応用を終わったばかりなので、この章は少々ハードですが、ぜひがんばってチャレンジしてください。

あなたまかせの割り込み

3–1■Intr 手続き

PC-9801 は CPU に 8086 系のチップを使っていますが、この CPU にはソフトウェア割り込み (インテラプト) 用の命令として INTR が用意されています。PC-9801 はモデルによって CPU が異なりますが、すべて 8086 系統のチップですから、以下の話ではすべて 16 ビット CPU である 8086 について説明します。

この割り込み命令は TURBO Pascal の Intr 手続きで使用することができます。ところが、TURBO Pascal のマニュアルでは割り込みとはなんなのかがわかっているひとを対象としているせいか、Intr の説明は実にあっさりとしているので、不慣れなひとには割り込みとは実際にどう使えるものなのか理解することはまずできないでしょう。

マニュアルの Intr 手続きの説明を読んでみても、その定義と呼び出し法が書いてあるだけです。考えてみればマニュアルとは規則書なのですから、どうしても応用例に限りがあるのはしかたがなさそうです。そこで、ここではまず割り込みの基礎のトレーニングから始め、後に行くにしたがって複雑なプログラムにチャレンジします。

さて、8086 ではソフトウェア割り込み用に物理アドレスの 00000H 番地から 003FFH 番地までの 1K バイトに、4 バイトずつ 00H、01H、02H、・・・FFH と 256 個の割り込み参照アドレスが書いてあります。この 4 バイトはオフセット 2 バイトとセグメント 2 バイトからなります。たとえば、Intr 手続きで 40 番 (16 進) の割り込みを実行すると、$40H \times 4 = 100H$ から 4 バイトで表されたアドレスにあるプログラムが実行されることになります。このアドレス表のことを割り込みベクタテーブルといいますが、この表がどうなっているのか、またそれをどう使うのかについては 3-2 節で説明します。

ここでは Intr 手続きによる割り込みの 18 番と 21 番の使い方をトレーニングします。18 番といっても正確には 16 進表記で 18H 番ですから、TURBO Pascal では$18 番になります。これが PC-9801 の ROM 内ルーチンである BIOS を動かす割り込みです。それに対して、21 番、つまり$21 番の割り込みはすでに第 1 章で MS-DOS のファンクション・コールとして取り上げました。

$18 と$21 番の割り込みは共通する部分がかなりあります。その理由は MS-DOS 上でも ROM の BIOS と同じ参照アドレスを用いているからです。言い換えれば、MS-DOS 独自の割り込みでなければ BIOS を用いる、つまり BIOS コールでも同じことができるといえます。さらに、MS-DOS のファンクション・コールに用意されていない ROM 内の BIOS 独自の機能でも BIOS コールを利用すれば実現可能ということです。しかし、BIOS コールを多用すると異なる機種のパソコンに TURBO Pascal で書いたプログラムを移植するときに苦労することになるので、その点はよく考えて使うべきです。つまり、どうしても BIOS を使わないと実現できない機能だけに限定して使用するのが正しい用法です。

あなたが持っているパソコンがPC-9801で、そのパソコン上でしか動作させないプログラムを開発するのならBIOSをどんどん使ってもかまいません。しかし、このエキスパート・マニュアルで中級クラスのプログラマーをめざすのなら、できるだけ移植性にも配慮をしたプログラム開発をするよう心がけてください。

さて、リスト3-1を見てください。このプログラムではBIOSコールによりブザーを鳴動させます。といってもそれほど大げさではなく、ただビーと鳴らし、またそれを止めるBEEP処理とキー入力をBIOSコールで行い、キー入力の画面表示をMS-DOSのファンクション・コールを用いて行います。ブザー鳴動はMS-DOSでも、エスケープ・シーケンスと呼ばれる特定の文字列(07H)を画面に送ることによっても実現可能です。しかし、ここでは練習のためにあえてやってみました。

■リスト3-1

```
 1: program LIST3_1;
 2:
 3: Uses
 4:        Crt,
 5:        Dos;
 6:
 7:
 8:   var
 9:     reg  :Registers;
10:
11: begin
12:   ClrScr;
13:
14:   reg.ax := $1700;
15:   Intr($18,reg);
16:
17:   WriteLn('どのキーでも押せば鳴動ストップします。');
18:
19:   reg.ax := $0000;
20:   Intr($18,reg);
21:
22:   Write('押したキーは:');
23:   reg.dx := reg.ax and $00ff;
24:   reg.ax := $0200;
25:   Intr($21,reg);
26:
27:   reg.ax := $1800;
28:   Intr($18,reg);
29: end.
```

プログラムの9行目はレジスタをアクセスするためにDosユニット内にあらかじめ定義されているタイプであるRegisters型として変数regを宣言しているところです。

もうわかっていることと思いますが、これを必ず行ってください。これにより reg でパソコンの内部レジスタがアクセスできるわけです。

BIOS コールとして Intr($18,reg) が 15 、20 、 28 行目に現れますが、A レジスタをセットするだけでファンクション・コールと同じ形式になります。プログラムの内容をちょっと見ておきましょう。

12 行目の ClrScr はもういうまでなく 5 行目で使用を宣言しておいた Crt ユニットの中の手続きで、画面をクリアするためのものです。14 行目でブザー鳴動のための BIOS コールの準備として A レジスタに $17 をセットしています。

19 、20 行目では押下されたキー文字を BIOS コールで取得し、23 〜 25 行目でそれを表示する MS-DOS ファンクション・コールをやっています。そして最後に 27 、8 行目で鳴動をストップする BIOS コールをやって終わります。以上の様子はプログラムと実行例を見れば簡単にわかるはずです。

これができれば一人前

3–2■割り込みベクタの書き換え

ベーシックなトレーニングはこのくらいにして、いよいよ本格的な割り込み処理をやってみましょう。ここでは例題として、ソフトウェア割り込みのひとつであるインタバルタイマの割り込みベクタを書き換え、画面右隅に時間を表示させるプログラムを作ります。そして、このプログラムを MS-DOS 上に常駐させ、常時画面に時刻表示が行われるようにします。

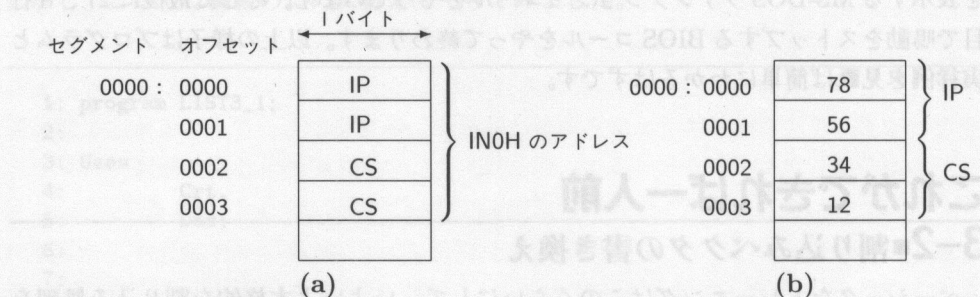
ここで開発する TURBO Pascal プログラムはかなり高級なテクニックを使いますので、がんばって理解するようにしてください。これが乗り切れればどんなプログラムでも作れるようになると思います。早くも第 3 章で山場がきたわけですが、しっかりトレーニングしてください。この節ではまず割り込みベクタと割り込みテーブルについて説明し、さらにインタバルタイマについても簡単に説明しておきます。

3–2–1■割り込みベクタと割り込みテーブル

あるプログラムを実行中に、ソフトウェア割り込みまたはハードウェア割り込みが発生した場合、CPU は現在実行しているプログラムを中断し、割り込み処理ルーチンに処理を移します。このときに、割り込み処理ルーチンがメモリ上のどこにあるかのアドレス情報を CPU が知らなければなりません。このアドレス情報が書かれているのが割り込みベクタテーブル、または簡単に割り込みテーブルと呼ばれるメモリ内の領域です。

CPU が 8086 系、つまり PC-9801 の CPU なら、割り込みベクタは割り込み処理ルーチンのセグメントアドレス (CS) とオフセットアドレス (IP) がペアになっています。こ

れを割り込みの0番であるINT 0Hを例にとって説明しましょう。HはHexadecimal(ヘキサデシマル)、つまり16進数であることの略号でしたね。

図3-1(a)を見てください。INT 0HのCSとIPがメモリ上の絶対アドレスの00000H番地から00003H番地までの4バイトを使って書かれています。たとえば、INT 0Hの割り込み処理ルーチンのアドレスがCS = 1234H、IP = 5678Hだとしましょう。このときメモリ上では図3-1(b)のような状態でこのアドレス情報が書いてあります。この状態でINT 0Hの割り込みが発生すると、この割り込みベクタの内容が参照され、コードセグメント・アドレスが1234Hでオフセット・アドレスが5678Hのメモリから始まる割り込み処理ルーチンへ処理が移されます。

■図3-1　INT0Hの割り込みベクタ例

つまり、CPUはこの割り込みにより、今まで処理していたプログラムの実行を中断し、こちらの割り込み処理ルーチンの実行を開始するのです。ここで重要なのは、割り込みが発生した場合、割り込みベクタ自体に処理が移されるのではなく、割り込みベクタに書かれているアドレスが参照され、そのアドレスのところに処理が移されるということです。

割り込みベクタはひとつの割り込み処理ルーチンのアドレスを4バイトで表しているのは前にも説明しました。図3-2を見てください。

割り込みの種類は256個まで可能ですから、全部で4 × 256 = 1024バイト分のメモリが割り込みベクタに必要です。これをメモリの絶対アドレスの00000H番地から5003FF番地までを使ってINT 0HからINT FFHまでの割り込みベクタを順番に並べています。この並びが割り込み処理がどこで行われるかの参照のための種の表(テーブル)になるので、割り込みベクタテーブルと呼ぶわけです。

この割り込みベクタはRAMメモリに書かれていますから、必要に応じて自由に書き改めることができます。とはいえ、なんでもかんでもでたらめに書き換えては意味がありません。MS-DOSを起動した時点で、MS-DOSが使用する割り込みベクタはMS-DOSによりセット(書き込み)されていますから、やたらに書き換えると動作が保証されなくなります。そこで、MS-DOSではINT 40H ~ 7FHをユーザー用としてユーザー、つ

まり私たちが自由に割り込み処理用として使えるよう開放しています。ここをどう書き換えても MS-DOS の動作は保証されます。

■図 3-2　割り込みベクタテーブル

それではここまでの知識で割り込み処理プログラムを作ってみましょう。リスト 3-2 は割り込みが発生するとブザーが鳴るプログラムです。同じ機能を実現するプログラムはすでにリスト 3-1 で紹介しましたが、比較するためにあえてベクタテーブルを書き換えて、割り込み処理ルーチンに制御を移動してみることとします。ここでは割り込みが 1 ～ 100 の正数を 7 で割り、余りが 2 となったときに発生するようにしました。適当な割り込み処理ルーチンを作り、ユーザー使用可能な割り込みベクタ INT 40H(00100H ～ 00102H 番地) のベクタに、その割り込み処理ルーチンのアドレスをセットし、割り込み処理をさせます。

■リスト 3-2

```
 1: program LIST3_2;
 2:
 3: Uses
 4:        Crt,
 5:        Dos;
 6:
 7: const
 8:        BELL=#$07;
 9:
10: var
11:        regs          :Registers;
12:        originaL_40h  :pointer;
13:        int_seg       :word;
14:        int_ofs       :word;
```

```
15:    i              :word;
16:
17: {*****************************************************}
18:
19: procedure Interrupts;
20: Interrupt;                                {割り込み処理用キーワード}
21:
22:   begin
23:      Write(BELL);
24:      WriteLn(i:3,'割り込みテスト中');
25:   end;
26:
27:
28: {*****************************************************}
29:
30: procedure Set_Vector;
31:
32:    begin
33:      GetIntVec($40,originaL_40h);          {オリジナルのベクターを退避}
34:
35:      SetIntVec($40,@Interrupts);           {割り込み手続きのベクターをセット}
36:    end;
37:
38:
39: {*****************************************************}
40:
41: procedure Reset_Vector;
42:
43:   begin
44:      SetIntVec($40,originaL_40h);          {ベクターを元に戻す}
45:   end;
46:
47: {*****************************************************}
48:
49: begin
50:    checkbreak := faLse;
51:
52:    CLrScr;
53:    Set_Vector;
54:
55:    for i:=1 to 100  do
56:     begin
57:
58:       if i mod 7 = 2 then
59:         begin
60:            regs.ds:=Dseg;
61:            Intr($40,regs);
62:         end;
63:     end;
```

```
64:
65:     Reset_Vector;
66:
67: end.
```

それではリスト3-2を見てみましょう。まず、19行目から始まる割り込み処理ルーチンであるInterrupts手続きを見てみましょう。本当は割り込みなのですから、その英語であるInterruptにしたいのですが、InterruptがTURBO Pascalの予約語なのでsを尻尾につけて規則にふれないようにしました。

20行目にあるInterruptは手続きInterruptsが割り込みサービスルーチン、つまり割り込みが発生してからどのような処理が行われるのかを記述したルーチンであることを示す予約語です。23行目で定数BELLを画面表示しているのはMS-DOSのエスケープ・シーケンスの画面送出によりブザー鳴動させるためです。実際には画面にBELLという文字が書かれるのではなく、8行目で定義された文字コードを画面出力すると、MS-DOSのエスケープ・シーケンスの約束によりブザーが鳴動します。24行目の出力はメイン部での繰り返しの第i回目ごとにブザー鳴動と割り込みのテスト中であることを画面に示すためのものです。

30行目からの手続きSet_Vectorはベクタテーブルを書き換えるためのものです。書き換えるからには、その前に何が書いてあったのかを保存して、このプログラムの実行が終わったら元の値に書き戻しておくべきです。このプログラムさえ走れば後はどうでもかまわないというのは、あまりお行儀(?)がよいプログラマーとはいえません。あなたがユーザーだとして、あるソフトを使ってから次のソフトに移ると原因不明の暴走が起こったりしたら腹が立つに決まっています。ですから、ベクタテーブルをいじったら必ず書き戻すように習慣付けておきましょう。

もっとも40H番の割り込みはMS-DOSのユーザーに解放していますから、ここをどう書き換えようともまったく影響ありません。とはいえ、よい習慣はどんなときにも守りたいものです。車がこないからといって赤信号を無視していると、いつの日か大きな事後を起こしかねません。自分で無駄だとわかっていても、よい習慣をくずすようなことはしないにこしたことはありません。

33行目のGetIntVecはDosユニットの手続きで、割り込みの40H番、つまりINT 40Hに書かれているベクタをoriginal_40hというポインタ変数にしまうためのものです。このようにあるデータを後で復帰させるために一時的に取っておくことを待避といいます。

35行目ではINT 40H番の割り込みが生じたときの飛び先、つまりサービスルーチンであるInterruptsのアドレスをセットするために、これまたDosユニットの中にある手続きSetIntVecを呼び出しています。@(アットマークと呼びます)はその後ろにくる変数のアドレスを示すTURBO Pascal特有の記号です。@InterrutsはInterrupts手続

きのエントリ・ポイント(入り口)のアドレスなのです。セグメントだオフセットだとめんどうなことを言わなくて済むのはありがたいことです。

41行目の手続きReset_Vectorによりベクタテーブルの復帰をやっています。以上の手続きInterruputs、Set_Vector、そしてReset_Vectorの3本がとても簡単に作れるのもTURBO Pascalに用意されているDosユニットのおかげです。これだけの処理を普通のコンピュータ言語でやろうとすると、あの手この手でアセンブラか機械語と連結させられますから、簡単に済むのは実にありがたい限りです。

メインプログラムは49行目から始まります。50行目で突然CheckBreakという変数が飛び出してきます。これはSTOPキーまたはCtrl-Cを押下してこのプログラムを中断させるかどうかを指定するための論理変数です。この変数はCrtユニット内で定義されているので、プログラム内で何の定義もなしにいきなり使えるのです。真(true)のときはSTOPキーまたはCtrl-Cで異常終了、つまりプログラムを実行中に強制停止できます。ここでは偽(false)にセットされていますから、強制停止はできません。

53行目でSet_Vectorによりベクタテーブルを設定し、55～63行目でiを1から100まで変化させ、それを7で割ったときに余りが2になるとき割り込みが59～62行目で発生するようになっています。iが2、9、16…のときにブザーが鳴動します。このプログラムがSet_Vectorで書き換えたベクタテーブルを65行目でReset_Vectorによりもとに戻しています。

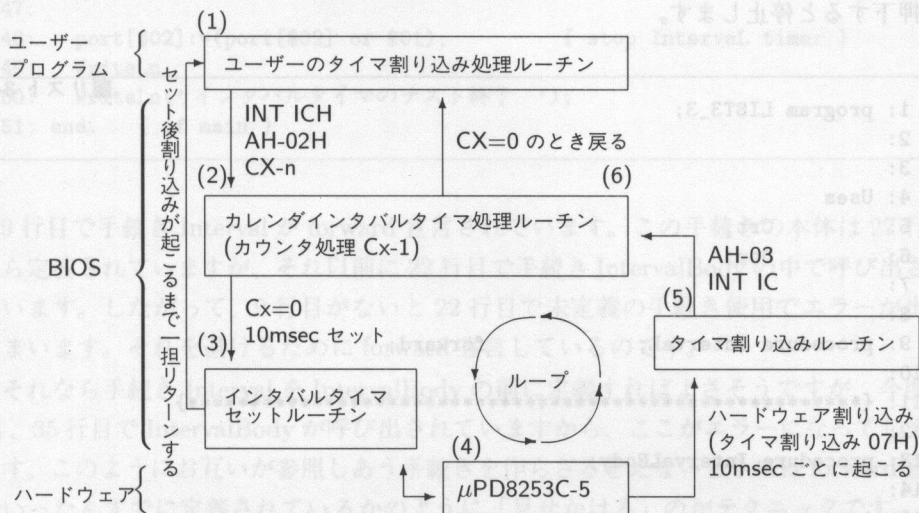
ここまでのトレーニングで、なんだこんなめんどうなことをしないでも簡単にブザー鳴動できるじゃないかと思うのが当然です。しかし、ここではどうベクタテーブルを書き換え、割り込みルーチンのアドレスをTURBO Pascalではどうやって引き渡しているのかを見ておいてください。これが次のステップのトレーニングに大切なのです。

3–2–2■インタバルタイマ

PC-9801には時間測定用のインタバルタイマというソフトウェア割り込みが用意されています。これと同等の機能は他のパソコンにも用意されているはずです。簡単にいうと、インタバルタイマとはある一定の時間間隔で自動的に割り込みを発生させるためのメカニズムのことです。後でそのインタバルタイマを利用して時間を測定するプログラムを作りますから、そのための予備知識としてインタバルタイマについて必要な基本的なことがらをここで説明しておきましょう。

まず、インタバルタイマを起動させるには、ユーザープログラムの中であらかじめインタバルタイマをどのように動かしたいのかのモード等をセットしておかなければなりません。それには、AH、つまりAレジスタの上位8ビットに02Hを、CX、つまりCレジスタ全体の16ビットにインタバル時間をそれぞれセットします。インタバル時間は10msec(1/100秒)単位です。たとえば、CX＝5とすると50msecごとに割り込みが発生します。さらに、ES(エクストラセグメント)のBXにインタバル時間がタイム

アウトになったときの戻り番地、つまり割り込み処理が終ったときに戻ってくる番地をセットします。CX ＝ 5 であれば、50msec 経過後どこの番地にあるプログラムを実行すればよいのかを ES：BX に書いておくわけです。これで INT 1CH を行うと CX レジスタにセットした時間で割り込みが発生します。

図 3-3 を見てください。インタバルタイマの動作原理は図の中の番号順に動作します。その番号にしたがって説明します。

■図 3-3　インタバルタイマ構成図

(1) ユーザー割り込みルーチンで各モードがセットされます。そして、INT 1CH が行われると割り込みベクタテーブルのベクタ番号 1CH の内容を参照して、カレンダ・インタバルタイマ処理ルーチンへ処理を移します。

(2) カレンダ・インタバルタイマ処理ルーチンでは、10msec ごとにタイマ割り込みが発生するようにインタバルタイマ・セット・ルーチンをセットし、処理をインタバルタイマ・セット・ルーチンに移します。

(3) インタバルタイマ・セット・ルーチンでは、μPD8253C-5 と呼ばれるインタバルタイマ用の IC に 10msec をセットします。

(4)μPD8253C-5 は 10msec ごとにタイマ割り込み INT 07H(ハードウェア割り込み) を発生し、タイマ割り込みルーチンに信号を送ります。

(5) タイマ割り込みルーチンでは AH に 03H をセットし、INT 1CH を行います。

(6) カレンダ・インタバルタイマ処理ルーチンへ処理が戻ってきます。カレンダ・インタバルタイマ処理ルーチンでは、CX レジスタから 1 を減らし、CX ＝ 0 になるとユーザー割り込みルーチンに戻ります。CX ≠ 0 であれば再び 10msec をインタバルタイマ・セット・ルーチンにセットします。

このように、(1) でセットされたCXレジスタの値が0になるまで (2)→(3)→(4)→(5)→(6)→(2)→···の順にループします。なお、CX に設定できる経過時間の範囲は 10(CX = 0001H) ~ 655360msec(約 11 分、CX = 0000H) です。

それではこのインタバルタイマ割り込みを使う TURBO Pascal プログラムを作ってみましょう。リスト 3-3 はインタバルタイマにより1秒ごとにブザーを鳴らし、同時に画面に"interval test"と表示するプログラムです。このプログラムはどれか任意のキーを押下すると停止します。

■リスト 3-3

```
 1: program LIST3_3;
 2:
 3:
 4: Uses
 5:         Crt,
 6:         Dos;
 7:
 8:
 9: procedure IntervaL;                  forward;
10:
11: {*****************************************************}
12:
13: procedure IntervaLBody;
14:
15: interrupt;
16:
17: const
18:        beLL=#$07;
19:
20: begin{IntervaLBody}
21:   WriteLn(beLL,'インタバルタイマのテスト中  ');
22:   IntervaL;
23: end; {IntervaLBody}
24:
25: {*****************************************************}
26:
27: procedure IntervaL;
28:  var
29:        regs:   Registers;
30:
31:  begin
32:    regs.ax:=$0200;
33:    regs.cx:=$0064;
34:    regs.es:=seg(IntervaLBody);
35:    regs.bx:=ofs(IntervaLBody);
36:
37:    Intr($1c,regs);
38:  end;  { of IntervaL }
```

```
39:
40: {*****************************************************}
41:
42: begin
43:   checkBreak := false;
44:   IntervaL;
45:
46:   repeat  untiL keypressed;                    { wait program end }
47:
48:   port[$02]:=(port[$02] or $01);               { stop IntervaL timer }
49:   WriteLn;
50:   WriteLn('インタバルタイマのテスト終了  ');
51: end.   { of main }
```

9行目で手続きIntervalがforward宣言されています。この手続きの本体は27行目から定義されていますが、それ以前に22行目で手続きIntervalBodyの中で呼び出されています。したがって、9行目がないと22行目で未定義の手続き使用でエラーが出てしまいます。それを避けるためにforward宣言しているのです。

それなら手続きIntervalをIntervalBodyの前に定義すればよさそうですが、今度は34、35行目でIntervalBodyが呼び出されていますから、ここがエラーになってしまいます。このようにお互いが参照しあう手続きを作らざるをえないときにはforward宣言でいったんすでに定義されているかのように「見せかける」のがテクニックです。

13～23行目の手続きIntervalBodyがやることは実質的にはブザー鳴動と画面に文字を表示するだけですから、引数を持っていません。ただ、22行目で27行目以下で定義されているインタバルタイマの設定手続きであるIntervalを呼び出しているところに注意してください。割り込みが発生するたびにある特定の動作(この場合はブザー鳴動と文字の画面表示)を行い、それが済んだらまたインタバルタイマに制御を引き渡すようにします。

27～38行目のInterval手続きではインタバルタイマの各モードを設定します。32行目ではAレジスタの上位8ビットに02Hを指定して、インタバルタイマのモード設定を指示します。1秒ごとの割り込みですから、33行目でCXに$64、つまり10進の100を設定して、割り込み時間を1秒(10msec × 100)にセットしています。そして、ES：BXには割り込み処理ルーチンである手続きIntervalBodyのセグメントアドレスとオフセットアドレスをセットします。これは割り込みから戻ってきたらどこへ行けばよいのかの番地になります。ここまで用意したところで37行目でベクタテーブルのベクタ番号1CHインタバルタイマ処理ルーチンに飛ばします。

メインプログラムは42行目からです。43行目でSTOPキーまたはCtrl-Cの押下を無効にしておきます。これによりプログラムが異常な終了をしてインタバルタイマが動いているままになっても悪影響を与えないようにするためです。46行目でキーが押さ

れるまで無限ループを設定します。このループを繰り返している間に1秒ごとにブザーが鳴動します。このループの中になにか処理を入れておくとその処理をしながら1秒ごとにブザーを鳴らすことができます。

48行目におかしな文がいきなり飛び出してきます。これはインタバルタイマの使用を終了するための処理です。これによりμPD8253C-5を停止し、図3-3のハードウェア割り込みの発生を終了しています。

3-2-3■時刻表示プログラム

それではいよいよインタバルタイマの割り込みベクタを書き換えて、スクリーンの右隅に現在の時刻を常時表示するプログラムを作りましょう。プログラムの説明に入る前にもう一度インタバルタイマの復習をかねて、図3-3を眺めながらどのようにプログラムを作ればよいのかあらすじを考えてみます。

図3-3で、インタバルタイマを起動させたときに、(4)のμPD8253C-5から10msecごとにハードウェア割り込みINT 07H(タイマ割り込み)を発生しています。つまり、これを利用すれば10msecという時間が手に入ります。むろん、インタバルタイマのCXレジスタに1をセットしても10msecはえられますが、ここでは割り込みベクタを書き換える方法を用いたプログラムを作るので、μPD8253C-5から10msecをえることにしました。

さて、INT 07Hの割り込みが発生するとタイマ割り込みルーチンへ処理が移されます。そして、AHレジスタに03Hをセットして、INT 1CHを行ってカレンダ・インタバルタイマ処理ルーチンへ処理がさらに移されます。そこで、このINT 07Hの割り込みベクタの内容を自作の割り込み処理ルーチンのアドレスに書き換え、10msecごとに自作の割り込み処理をさせることにします。この自作の割り込み処理ルーチンの中にカウンタを作り、時間を計算表示させるようにすれば、うまくいきそうです。

具体的な動作としては、図3-3の(1)でインタバルタイマをセットし、INT 1CHを行い処理を(2) → (3) → (4)と移します。そして、(5)で自作の割り込み処理ルーチンで時刻表示を行い、インタバルタイマの各モードをセットし、(5)のルーチンからINT 1CHを行います。こうすれば(5) → (6) → (2) → (3) → (4) → (5) →···のループで時刻を表示することができるのです。

■リスト3-4

```
1: {$M 1024,0,0 }
2:
3: program LIST3_4;
4:
5: Uses
6:   Dos;
7:
8: procedure IntervaLTimer ;forward;
```

```
 9:
10: var
11:      h, m, s, s100   : word;        { s100 is dumy }
12:
13: {$F+}
14: procedure InterruptBody;Interrupt;
15:
16: begin  { of InterruptBody }
17:   InLine($fa);      { CLI … 割り込み禁止 }
18:
19:   { 時間カウント処理 }{ 割り込み処理ルーチン内では GetTime を使えないため }
20:
21:   inc(s);{ s := s +1 }
22:
23:   if s >= 60 then
24:   begin
25:     s := 0;
26:     inc(m);
27:   end;
28:
29:   if m >= 60 then
30:   begin
31:     m := 0;
32:     inc(h);
33:   end;
34:
35:   if h >= 24 then
36:     h := 0;
37:
38:   { dispLay time on screen }
39:
40:   mem[$a000:$0088]:= $30 +(h div 10);
41:   mem[$a000:$008a]:= $30 +(h mod 10);  { hours }
42:   mem[$a000:$008c]:= $3a;  { 3A is ':' }
43:   mem[$a000:$008e]:= $30 +(m div 10);
44:   mem[$a000:$0090]:= $30 +(m mod 10);     { minutes }
45:   mem[$a000:$0092]:= $3a;  { 3A is ':' }
46:   mem[$a000:$0094]:= $30 +(s div 10);
47:   mem[$a000:$0096]:= $30 +(s mod 10);    { seconds }
48:
49:   IntervaLTimer;
50:
51:   InLine($fb);          { STI … 割り込み許可 }
52: end; { of InterrutBody }
53: {$F-}
54:
55:
56: procedure IntervaLTimer;
57: { インターバルタイマーの設定 }
```

```
58: var
59:     dos_reg   : Registers;
60:
61: begin { of IntervaLTimer }
62:    dos_reg.ax := $0200;
63:    dos_reg.cx := $0064;   { 1sec }
64:    dos_reg.es := Seg(InterruptBody);
65:    dos_reg.bx := Ofs(InterruptBody);
66:
67:    intr($1c, dos_reg);
68: end; { of IntervaLTimer }
69:
70:
71: begin
72:   GetTime(h,m,s,s100);   { function read time }
73:
74:   IntervaLTimer;
75:
76:   keep(0);             { TRS program }
77: end. { of main program }
```

こうして作ったプログラムがリスト3-4です。1～3行目はコンパイラ指令です。1行目の$M指令はメモリの割当サイズを指定するものです。一般的な形式は

{$M　スタックサイズ，ヒープ最小サイズ，ヒープ最大サイズ}

です。このプログラムでは作業領域としてスタックを使用します。しかし、必要メモリ容量としてどれだけとればよいのかはマニュアルのどこにも書いてありません。そこでぼくは試行錯誤的に最初は5～6Kバイトほどと大きめにとっておき、次第に減らしてこの値にしました。ヒープは使わないのでメモリ確保する必要がありません。

13行目に{$F＋}のコンパイル指令があります。＋を指定すると必ずFARコール、－だとTURBO Pascalが適当に決めてFARまたはNEARコールが手続きの呼び出しに使われます。そういわれて、ああそうですかと理解できるのはかなりの上級プログラマです。ちょっと変わったことをやろうとするときには安全策をとって＋を指定しておけば間違いありません。ここではその方針で{$F＋}としています。

8行目で割り込みの部分をforward宣言しておきます。10、11行目で宣言されている変数はそれぞれhが時、mが分、sが秒を表します。s100は100分の1秒を表しますが、あとでメインプログラムの72行目でGetTimeの引数に使われています。しかし、PC-9801では100分の1秒はサポートされていないので、s100は単なる意味のない引数になります。つまり、引数の数合わせのためだけの変数です。

14行目から割り込み処理用の手続きです。17行目で8086の割り込み禁止命令CLIを直接機械語で行っています。この命令はTURBO Pascalに存在しないのでInlineを

使用しました。これは割り込み処理中に別の割り込みが発生するのを防止するためのものです。このあたりの話になると、話がよく見えなくなる (?) かも知れませんが、割り込み処理の「定石」としてほかのプログラムを流用するときにこのようにするものだと思ってください。

21 行目で 1 秒進めています。割り込み処理ルーチン内では GetTime を使用できませんから、時刻の進め方は自分で作らねばなりません。その理由は GetTime は Dos ユニットの関数ですから、コンパイル時に dos.tpu をこのプログラムに結合します。そしてプログラムの終了とともに dos.tpu の部分もメモリからなくなります。一方、このプログラムは割り込み処理プログラムをメモリに常駐させて終わりますから、その中でなくなってしまうものから手に入れた GetTime を使うわけにはいかないのです。

24 〜 37 行目で 24 時間表示用のデータを計算します。40 〜 47 行目では VRAM に直接データを書き込んでいます。ここで$30 は文字コードの 0 を表しています。

49 行目で forward 宣言されていた IntervalTimer を呼び出します。後で出てくるように、この手続きでインタバルタイマの設定をします。ここまでで割り込み処理が終わりますから、51 行目で STI 命令を出して割り込みの許可をします。これは 17 行目と対になって割り込み処理の定石を構成しています。53 行目で TURBO Pascal の手続き呼び出し方式に戻しておきます。これも 13 行目と対になります。

56 行目からはインタバルタイマの設定手続きです。これはもうすでに前のプログラムで説明済みのテクニックです。

メインプログラムは 71 行目から始まります。72 行目でまずカレンダ時計の時刻を読み込みます。これを基準としてインタバルタイマから発生する 1 秒ごとの割り込みで時刻表示を進めて行きます。74 行目でインタバルタイマを設定します。76 行目で Keep という見慣れない手続きが出てきます。この手続きはプログラムをメモリ内に TSR(常駐) させるためのものです。TSR とは Termenate Stay Resident 、常駐終了の略号です。$M コンパイラ指定なしにこれをやると、プログラム終了後に MS-DOS の command.com がメモリに戻ろうとしても空き場所がなくなり、リセットする以外に回復できない致命的なエラーが発生します。

しかし、TURBO Pascal には常駐したプログラムをメモリから追い出す、つまり解放する手続きはありません。したがって、このプログラムが終了して画面の右上に時刻が表示されるようになると、メモリ使用可能量は減少します。試しにこのプログラムを走らせる前と後で chkdsk コマンドを使用して調べてみてください。Keep の引数はプログラムの終了コードなのですが、ここでは常駐終了するのであまり意味を持ちません。ただ、プログラムの開発途中では常駐終了がどんな終わり方をしたのかをチェックするには役にたつでしょう。

これですべて完成したのですが、ここで紹介したリスト 3-4 のプログラムは他の目的に簡単に改造することができます。つまり、このプログラムのパターンを眺めてみると

次のようになります。

```
{$M+}
program ......;
uses Dos;
procedure IntervalTimer : forward;
{$F+}
procedure InterruptBody:interrupt;
  begin
    Inline($fa);
    ..............
    IntervalTimer;
    Inline($fb);
  end;
{$F-}
procedure IntervalTimer;
..............
begin
  メインプログラム
  ..............
  IntervalTimer;
  Keep(0);
end.
```

この定石を使えば割り込み処理ルーチンのInterruptBodyの中身を書き換えたり、メインプログラムで設定する割り込みの種類を変えてやればいくらでも改造できます。たとえば、毎正時にブザーを鳴らすことなど簡単にできます。また、実験用にTURBO Pascalでプログラムを書いて、測定値をある時間ごとに読み込んで処理をし、しかもその測定中に他のプログラムを実行させたりするという離れ技もこれで可能となります。

4 地味な世界を見直そう

第3章ではかなり高度なTURBO Pascalによる割り込みをやってみました。プログラムの中でいったい何をやっているのか調べるのに少々疲れてきましたか。この章ではキーボード関係のテクニックをトレーニングします。前の割り込みに比べるとぐっとやさしいプログラムばかりですから、気楽にチャレンジしてみてください。

キーボードは入力装置の基本です。最近はウィンドウ環境でマウスを積極的に使用するソフトウェアが増えてきましたが、やはりパソコンに文字情報を入力するとなると、キーボードが主役になります。ここで紹介するプログラムはどれも簡単なものばかりですが、実用的に十分価値のあるプログラムを作るためのヒントになるはずです。

キーボード関係をいろいろいじり回すにはデバイス・ドライバを作らなければならないのですが、中級プログラマーへのトレーニングとしてはTURBO Pascalで簡単にプログラミングできる範囲に絞ってあります。とはいえ、ここで紹介するサンプル・プログラムはどれも応用範囲が広いものですから、例題を入力して動作を確認しながらテクニックを勉強しましょう。

BASICに負けそう

4–1■INKEY$

BASICで作ったプログラムを読みやすくするためにTURBO Pascalに移植することはよくあります。プログラミングの勉強をBASICでまず始めたひとがほとんどでしょう。BASICのよさはなんといっても思いつくままにプログラムを書けるところにあります。厄介なプログラムの書き方、つまり書法にとらわれることなく、のびのびとプログラムを書けるのは魅力です。

ところが、あまりのびのび書いた結果、あとでプログラムを改良しようとしたり、ほかのひとにプログラムを読んでもらおうとしたときに困ったことがおこります。はてプログラムの論理(ロジック)はどうだったのかとか、ここで一時的に使われる変数はなんだったのかなどということになります。こうした問題はいろいろな参考書に載っているBASICのプログラムを理解するときにもおこります。

また、BASICで書かれたプログラムをTURBO Pascalに移植するときに、BASIC特有の機能をどう実現するかという問題もあります。これが原因で、いままで自分で開発したBASICをTURBO Pascalの中に積極的に利用できないこともよくあるようです。TURBO PascalよりBASICのほうがグラフィックスが充実していたので、きれいな表示を必要とするプログラムをBASICで作らざるをえないことがありました。

しかし、TURBO Pascalもバージョンアップを重ねてグラフィックス機能はBASICと遜色がないどころか、それをしのぐほどになりました。こうなるといままで開発したり参考書に載っているBASICプログラムをTURBO Pascalに移植したくなるのが人情(?)です。そこで出てくるのがTURBO Pascalに用意されていないBASIC特有の命令をどう移植するかという問題です。

その中でもBASICのINKEY$関数はTURBO Pascalに移植するとなると少々厄介です。リスト4-1を見てください。INKEYはキーボードで押下したときそのキー文字を返す関数です。短いプログラムですからリストの書き込みを参考にすれば内容は理解できるはずですが、簡単に説明しておきます。

■リスト4-1

```
 1: program LIST4_1;
 2:
 3: Uses Crt;
 4:
 5: function Inkey:char;
 6: {1文字取得関数}
 7: var
 8:     in_char : char;
 9:
10: begin
11:     Inkey := #00;
12:     if KeyPressed then
13:     begin
14:       DeLay(0);{機種により調整必要}
15:       Inkey := Readkey;
16:     end;
17: end;
18: {------------------------------------}
19: {Main Program}
20: begin
21:   repeat untiL Inkey = ' ';
22:   {スペースが押されるまで無限ループ}
23: end.
```

3行目でCrtユニットを使用しているのは後でこのユニットに属する関数、KeyPressedとReadKeyを使用するためです。5行目でこれから開発する1文字取得関数をInkeyとして文字関数定義します。

この関数の本体は10行目からです。11行目で初期値としてこの関数の何も押されないときの値をヌル文字に設定しておきます。12行目でキー押下があったかどうかを検出します。14行目のDelay関数はパソコンの機種によっては0以外の適当な値に設定しないと、Keypressed関数がハードウェアのタイミングの関係で動作しないことがあるので、そのためのものです。「適当な値」は試行錯誤で決定します。

このリストを見ると、「なんだ、こんなことならただReadKey関数だけで同じことができるんじゃないか」と思ったのではありませんか。しかし、ここで注意しておきたいのは、この関数はASCIIコード対してのみ有効だということです。ATOKやVJEなどの日本語フロントエンドプロセッサで入力される日本語の2バイトコードはどうしたらよいのでしょうか。ASCIIコードは1バイトで1文字に対応するのに対して、MS-DOSの日本語コード、つまりシフトJISコードは2バイトで1文字に対応するのですから、ちょっと厄介です。

■リスト 4-2

```
 1: program LIST4_2;
 2:
 3: Uses Crt,Dos;
 4:
 5: type
 6:   Str2 = string[2];
 7:
 8: var
 9:   command   : char;
10:   character :  Str2;
11:
12: function Inkey(var KeyChar : Char):booLean;
13: {1文字取得関数}
14: var
15:   DosRec : Registers;{レジスタアクセス用}
16:
17: begin
18:   if KeyPressed then
19:     begin
20:       DosRec.ax := $0800;
21:       Msdos(DosRec);
22:       keychar := Chr(Lo(DosRec.ax));
23:       Inkey := TRUE;
24:     end
25:    eLse
26:       Inkey :=FALSE;
27: end;
28: {-----------------------------------------------}
29: {Main Program}
30: begin
31:   CheckBreak := FALSE;
```

```
32:   CLrScr;
33:   WriteLn('INKEY ノ デモ. ドノキーデモオシテクダサイ.',
34:                          '<ESC>キーデ トマリマス.');
35:   WriteLn;
36:   command := #00;{取得文字初期化}
37:
38:   repeat
39:   {ESCキー (コード27) が押されるまでループする}
40:     if InKey(command) then
41:       begin
42:         Character := command;
43:         if Ord(command) > $80 then
44:         {シフトJISコード}
45:           begin
46:             Write('カンジ コード デス. コードハ',Ord(command));
47:             if InKey(command) then
48:                Write(':',Ord(command));
49:             WriteLn(' ',Concat(character,command));
50:           end
51:         eLse
52:         {ASCIIコード}
53:           WriteLn('アスキー コード デス. コードハ',Ord(command),
54:                                               ' ',command);
55:       if command <>#27 then WriteLn('ドノキーデモオシテクダサイ.',
56:                                        '<ESC>キーデ トマリマス.');
57:       WriteLn;
58:     end;
59:   untiL command = #27;
60: end.
```

リスト 4-2 の Inkey 関数は複数のバイトが一度に入力されても、きちんとそれを判別してコードを返すように改良したものです。12 行目の Inkey 関数はキー入力があると値が TRUE となり、その入力文字を引数として返す関数です。19 ～ 24 行目で第 2 章で勉強した MS-DOS のファンクション・コールを使用して押下されたキーのコードを取り出しています。

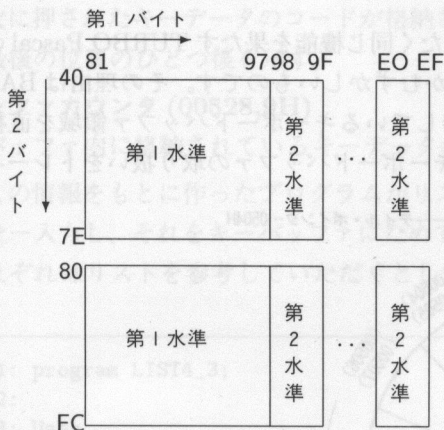
20 行目の A レジスタ設定はエコーなしのキーボード入力のファンクション・コールです。エコーとはエコーバックともいいますが、キー入力をそのまま画面に表示することです。このプログラムでは必要ないのでエコーは表示しません。つまり、何が押されたのかが自動的に表示することをやりません。

メインプログラムは 29 行目からです。このプログラムはエスケープ・キーを入力されるまで動きます。押下文字は変数 command に格納されますから、36 行目でヌル文字に初期化しておきます。

38 ～ 59 行目で取得文字が ESC キー (コード 27) になるまで繰り返します。キーのど

れかが押下されるとその文字をとりあえず42行目でcharacterという2文字長の文字変数に格納します。これでcharacterの1文字目はcommandと同じになります。

43行目で入力文字の1文字目がASCIIコードで80Hを越えたらその文字はシフトJISコードの1文字目であると判定しています。

シフトJISコードでは2バイトで1文字の日本語(漢字)を表すようにしています。シフトJISコードは図4-1のようなバイトの組み合せになっていますから、未定義のコードの1バイト目が80H以上のものはたくさんあります。したがって、1バイト目だけでシフトJISコード入力の判定をするのには問題があるのですが、ここではプログラムを簡単にして1バイト目だけで判定させています。もちろん、1バイト目と2バイト目を判断して、シフトJISコードなのか未定義コードなのかも分けて表示することもそれほど難しくないでしょう。

■図4-1 シフトＪＩＳの構成

46行目でまずシフトJISコードの1バイト目の文字コードを表示します。47行目のInkeyの2度目の呼び出しは単にシフトJISコードの2バイト目を読み出すためだけのものです。48行目で2文字目のコードを:(コロン)の後ろに表示し、49行目で2バイト目として得られたcommandをcharacterにつなげて画面表示します。こうすると漢字が表示されます。

51 ~ 58行目は入力文字がアスキーコードの場合です。このときはそのまま画面にコードと文字を表示させるだけです。55行目はESCキーが押されたらプログラムが終了するので、そのための準備を行います。#27はエスケープ・キーのコードですから、ESCの押下により繰り返しを終了します。

ここで紹介したテクニックを応用すると、アスキーコード入力とシフトJISコード入力が混在するようなプログラム開発に便利です。たとえば次のようにユーザーに入力させることがよくありますが、そのときに日本語入力しているといちいち半角、つまりCtrl-XFERなどを押下してASCIIコード入力に切り替えないと動作しないプログラム

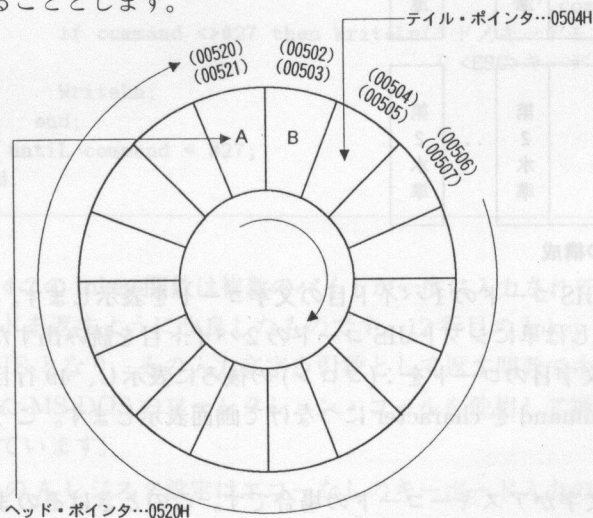
がよくあります。

```
古いデータを捨てますか？　Y(yes) or N(no)
```

ぼくたち素人が作ったプログラムならまだしも、売りものになっている市販ソフトでもそんなことがあるのだから驚きです。ここで解説したようなテクニックを使えば、ユーザーに厄介なことをさせない親切プログラムが簡単に実現できます。

ここがキー入力の溜り場

4–2■キーボードバッファ

4-1節ではBASICのINKEY$関数とまったく同じ機能を果たすTURBO Pascalの手続きを作ろうとしました。しかし、なかなかむずかしいものです。その理由はBASICのINKEY$関数はキー入力を一時的に保持しているキーボードバッファ領域を直接参照に行っているためです。そこでここではキーボードバッファの取り扱いをトレーニングしてみることとします。

■図 4-2 キーバッファ概念図

さて、キーボードバッファといっても特別なメモリがあるわけではありません。PC-9801ではシステム共通のRAMメモリ中のセグメント0000H、オフセット0502H～0521Hの32バイトがそれに割り当てられています。格納されるキーコードは押下されたキーの位置を示す1バイトと、シフト制御を含む1バイトの計2バイトで構成されています。したがって、バッファ内には最大16文字まで蓄えることができます。

この32バイトは図4-2のように、環状のバッファにエンドレスとなっています。つまり、00502Hから始まって00521Hまでがつながっています。とはいっても、実物がこうなっているのではありません。あくまでも概念としてこうなっているだけです。

このような環状の構造になっていると、読み出すべき文字列の最初の文字と終わりの文字がどれだかわからなくなってしまいます。そこで、オフセット00524～529Hの6バイトに、このキーボードバッファを管理するための各種ポインタを格納しています。つまり、キーボードバッファからの読み出しはこれらポインタを参照しながら行われるのです。ポインタは次の3つです。

ヘッドポインタ (00524,5H)

次に読み出されるキーデータの格納位置を示す。

テイルポインタ (00526,7H)

次に押されたキーデータのコードが格納されるアドレスを示す。つまり、格納文字列の最後の位置のひとつ後を示す。

バッファカウンタ (00528,9H)

バッファ内に格納されているキーデータの個数を記憶する。

この情報をもとに作ったプログラムがリスト4-3です。このプログラムでは適当な回数キー入力し、それをキーバッファにためておいてその内容を画面表示します。変数のそれぞれはリストを参考していただくとしてプログラムの内容を見てみましょう。

■リスト4-3

```
 1: program LIST4_3;
 2:
 3: Uses
 4:   Crt, Dos;
 5:
 6: var
 7:   data : array[1..32] of integer;{キーバッファ}
 8:   regs : Registers;
 9:   a,an,n,ec : char;              {作業用文字変数}
10:   b,c,d,e,f,i :integer;          {作業用整数変数}
11:   taiL,head,kba : integer;       {キーバッファポインタ}
12:   dumy : integer;
13:
14: begin
15:   repeat
16:
17:     regs.ax:=$0300;
18:     Intr($18,regs);                { キーバッファ初期化 }
19:
20:     kba:=$502;                     { キーバッファ・アドレス }
21:     CLrScr;
22:     GoToXY(15,10);
```

```
23:     WriteLn('適当なキーをどれでも押してください。');
24:
25:     GoToXY(15,12);
26:     DeLay(10000);              {この間にキーを押す}
27:
28:     for i:=1 to 32 do          { キーバッファから配列への読み込み }
29:     begin
30:       data[i]:=mem[0000:kba];
31:       Inc(kba);
32:     end;
33:
34:     head:=memw[0000:$0524];          { ヘッドポインタ・アドレス }
35:     taiL:=memw[0000:$0526];          { テイルポインタ・アドレス }
36:
37:     CLrScr;
38:     GoToXY(15,8);
39:     WriteLn('希望する表示の番号を選んで下さい！');
40:     GoToXY(15,10);
41:     WriteLn('1. キーバッファの一部を表示');
42:     GoToXY(15,11);
43:     WriteLn('2. すべてのキーバッファを表示');
44:     GoToXY(15,13);
45:     Write('どちらですか？:');
46:
47:     repeat                           { 1,2の入力 }
48:       GoToXY(24,13);
49:       n := ReadKey;
50:     untiL (n = '1') or (n = '2');
51:
52:     case n of
53:
54:     '1' :  begin
55:              repeat
56:                repeat
57:                  CLrScr;
58:                  WriteLn('head : ',head);
59:                  WriteLn('taiL : ',taiL);
60:                  Write('何番目の内容ですか 1 - 9 ? : ');
61:
62:                  ec := ReadKey;     { ここはread(ec); ではダメ！ }
63:                untiL ('1'$<$=ec) and (ec$<$='9');
64:
65:                vaL(ec, e, dumy);   { 文字数字変換 $\cdots$ e := ec -$30 }
66:
67:                WriteLn;
68:                i := e$*$2-1;
69:
70:                GoToXY(7,7);
71:                Write('Number  ','Character  ','ASCII code  ',
```

```
 72:                                    'Key Code  ');
 73:            GoToXY(8,9);
 74:            Write('No ',e);
 75:
 76:            GoToXY(19,9);
 77:
 78:            if data[i]=32 then
 79:              Write('sp')
 80:            eLse if data[i]=27 then
 81:              Write('Esc')
 82:            eLse if data[i]=13 then
 83:              Write('CR')
 84:            eLse
 85:            begin
 86:              GoToXY(19,9);
 87:              WriteLn(Chr(data[i]));
 88:            end;
 89:
 90:            GoToXY(29,9);
 91:            WriteLn('[',data[i],']');
 92:
 93:            GoToXY(40,9);
 94:            WriteLn('[',data[i+1],']');
 95:            WriteLn;
 96:
 97:            GoToXY(9,12);
 98:            Write('もう一度キーバッファを表示しますか？  y/n : ');
 99:
100:            an := ReadKey
101:          untiL (an ='n') or (an ='N');
102:
103:        WriteLn;
104:      end; { of '1' }
105:
106:    '2': begin
107:        CLrScr;
108:        GoToXY(5,4);
109:        WriteLn('head : ',head);
110:        GoToXY(5,5);
111:        WriteLn('taiL : ',taiL);
112:        c:=9;
113:        GoToXY(7,7);
114:        Write('Number   ','Character   ','ASCII code  ',
115:                                    'Key Code  ');
116:        for e:=1 to 16 do
117:          begin
118:            i :=e$*$2-1;
119:            GoToXY(8,c);
120:            Write('No ',e);
```

```
121:           GoToXY(19,c);
122:
123:           if data[i]=32 then
124:             Write('sp')
125:           eLse if data[i]=27 then
126:             Write('Esc')
127:           eLse if data[i]=13 then
128:             Write('CR')
129:           eLse
130:           begin
131:             GoToXY(19,c);
132:             Write(Chr(data[i]));
133:           end;
134:
135:           GoToXY(29,c);
136:           Write('[',data[i],']');
137:           GoToXY(40,c);
138:           WriteLn('[',data[i+1],']');
139:           inc(c);
140:         end;
141:       end;
142:     end; { of case }
143:
144:     Write('          もう一度始めからやりますか？  y/n : ');
145:     an := ReadKey;
146:   untiL (an='n') or (an='N');
147: end.
```

このプログラムは 15 行目の repeat と 146 行目の until の間を繰り返すようになっています。17 行目はキーボードバッファを初期化するための BIOS コールです。この BIOS コールによってキーボードバッファはむろんのこと、あらゆるキーボードに関する情報がクリアされます。26 行目はキー入力するために 10 秒間待つところです。もっともパソコンの機種によっては Delay 関数の値を書き換えなければなりません。

ここでキーをどれでもガチャガチャと押してもらいます。むろん画面には押したキーの文字、つまりエコーは表示されませんがキーのコードはキーバッファに蓄えられます。Delay の代わりに Read(a) や ReadLn(a) でキー入力を読ませると、ヘッドとテイルのポインタの値が同じになりますから、確かめてください。つまり、このように変更すると入力は CR で切られますから、次の入力用にヘッドとテイルを同じにして、「まだ入力がない」状態でキー入力を待つわけです。

34 、35 行目でヘッドとテイルポインタをメモリを直接アクセスして取り出しています。この 2 つの差がキーを押した回数ということになります。これ以降のプログラムの残りの部分はすべてのキーバッファを表示する部分と、指定された位置のデータを表示する部分でできています。ここはいわば結果の表示だけの問題で、キーバッファの使用

法とは関係ありません。リスト内の書き込みを参照してもらえば、説明は必要ないでしょう。ただ、68行目で配列data［i］のiを奇数にしているところだけ注意してください。これは奇数側にアスキーコードが、偶数側にキーコードが入っているからです。

このプログラムを走らせたら日本語フロントエンドプロセッサで漢字を入力してみてください。たとえば、ローマ字漢字変換を使ったATOK6だと、「亜」を入力するにはAをタイプしてスペースキーで変換します。これでリターンキーを押下して入力してみるとキーバッファにはアスキー文字のAとスペース、つまりキー入力そのものが入っています。「亜」のシフトJISコードがキーバッファに入れられわけではないことがわかります。このことは日本語フロントエンドプロセッサがMS-DOSのデバイス・ドライバを使用して日本語入力を行っていることによります。こう見てくるとアスキーコードだけならまだしも、漢字入力まで正確に取り扱えるINKEY$関数を作るのは簡単ではないということになります。

とはいえ、これでキーバッファからの押下キーコードの読み取り方がわかったでしょう。それを利用すればおもしろいプログラムが開発できそうです。ぜひいろいろ試してみてください。

ユニットをつくる

4-3■制限時間付きキー入力

こんどはちょっと趣向(?)を変えて、ある指定時間内にキー入力があったかどうかを調べるプログラムを作ってみましょう。むろん、ただ調べるだけではなく入力があったときにはそれがなんだったかを取得できなくては意味がありません。

ここではトレーニングをかねて時間制限付きの関数を自作ユニットとして開発します。まず、大きなプログラムを作るための基礎知識として大切なユニット機能について、復習をかねて解説することにします。

ユニットの基本的構造と組み立て方は次のようになっています。

```
unit　ユニット名　(1)
interface　(2)
uses　使用されるユニット名　(3)
パブリック宣言　(4)
implementation　(5)
uses　使用されるユニット名　(6)
プライベート宣言　(7)
手続き、関数の記述　(8)
begin　(9)
```

初期化記述 (10)

end. (11)

各部分に付けた番号にしたがって説明しておきます。

(1)unit ユニット名

ここにはそのユニットにつける名称を書きます。Pascal の program 宣言と同じものと考えてよいでしょう。混乱を避ける意味で、そのユニットを記述した TURBO Pascal のソースコード・ファイルと同じ名称にするのが普通です。たとえば、edit.pas というソースコード・ファイル名でユニットを作るのならば、ユニット名も edit としておきます。

(2)interface

これに引き続いてこのユニットのインタフェース部が宣言されます。いったんコンパイルが終了すればこのユニットはどのプログラムからもその名前を uses 分で宣言すれば使えます。しかし、プログラムからはこのインタフェース部で宣言されている部分しかユニットの内部を知ることはできません。具体的には外部からはユニットを作った人が使用者に対して公開 (パブリック) にする必要のある定数や変数、データ型や手続き、関数しか知ることができないのです。このように意図的に外部からユニットの内部を隠せますから、非常に便利で商品性を持つようなユニットを開発することも可能になります。

(3)uses 使用されるユニット名

あるユニットが他のユニットを使用するときにはここに宣言します。ここで宣言された他のユニットも当然インタフェース部を持っていて、定数、変数、データ型を公開しています。それがこのユニットのインタフェース部で宣言されているのと同じことになります。

(4) パブリック宣言

ユニットの外部から参照できる部分を宣言します。つまり、ここがインタフェース部の実体で、定数、変数、データ型、関数、手続きを宣言するところです。

(5)implementation

ここから下にそのユニットの本体を記述します。この記述は外部からは見えない、つまり参照することはできない非公開 (プライベート) の部分になります。

(6)uses 使用されるユニット名

このユニットでプライベートに使用されるユニット名を宣言します。外部からはこのユニットが他にどのようなユニットを使用しているのかを知ることはできません。

(7) プライベート宣言

ここにはそのユニット特有の定数、変数、データ型を宣言します。これも当然非公開ですから、ユニットの外からどのようなものが宣言されているのか知ることはできません。

(8) 手続き、関数の記述

ここにはインタフェース部で外部から参照可能とした関数、手続きを記述することは当然ですが、さらにこのユニット内だけで使用されるまったく非公開の関数、手続きを記述することもあります。インタフェース部で宣言された関数、手続きはその名称を宣言するだけでかまいません。たとえば、インタフェース部に、

```
function Triangle(w,h : real);
procedure AngleCoord(var ang1,ang2 : integer);
```

と宣言してあれば、ここでは単に、

```
function Triangle;
procedure AngleCoord;
```

とするだけでよいのです。

当然のことですが、このユニット内だけで使用されるまったく非公開、つまりインタフェース部に宣言のない関数、手続きはこのような省略はできません。普通の関数、手続きの規則に従って function あるいは procedure 文から始めてフルに記述しなければなりません。

(9)begin

ユニットが実際に使用されるに先立ってある特定の変数を初期化しておきたいときに必要です。ユニットは本質的には関数と手続きの集合ですから、実行される以前に何らかの初期化を行うことは不可能です。それをどうしてもしたいときにこれを利用するのです。

(10) 初期化記述

ここに実際の初期化の内容を記述します。書き方は普通の Pascal のプログラムと変わりません。このユニットを使用しているプログラム本体は他にもユニットを使用している可能性があります。そのときにはプログラム本体の uses で宣言された順序で各ユニットの初期化が行われます。この初期化はプログラム本体の実行に先立って行われますから、十分注意して作らないと原因不明のバグが生まれたりします。

(11)end.

この end は (9) の begin に対応しているように思いがちですが、実は (5) の implementation に対応しているもので、プライベート部分の記述の終了を示すものです。したがって、初期化の部分がない場合には、

```
unit Reidai;
...
implementation
....
関数、手続きの記述
...
end;
end.
```

となります。Pascal の大原則では begin と end は 1 対 1 に対応して、複数の文 (ステートメント) をその対ではさむことでしたから、最後の end は字余り (?) になり、おかしな印象を与えます。もっとも、本来の Pascal にはユニットなどという概念はないのですから end の意味も崩れて当然かも知れません。ここまでの知識でリスト 4-4A を見てください。

■リスト 4-4A

```
 1: unit LIST4_4A;
 2:
 3: {時間制限付き読み込みユニット}
 4:
 5: interface
 6:
 7: procedure TimedRead( var S : String; TimeLim : integer);
 8:
 9:
10: impLementation
11:
12: Uses Dos;
13:
14: procedure TimedRead( var S : String; TimeLim : integer);
15:
16: { TimeLim は時間制限の秒数。1 から 59 までの値をとり、その秒数
17: 間入力を待ち、この間入力がなかった場合、入力値として NuL を返す。}
18:
19: const
20:    CR  = ^M;{改行のコード CtrL-M}
21:    NUL = ^@;{無入力のコード CtrL-@}
22:
23: var
24:    GetOut : booLean;
25:    Now,            {現在時刻}
26:    TimeUp : integer; {終了時刻}
27:    Regs : Registers;
28:    ch : char;
29:
30: function ReadDosSecs : integer;
31: { システムクロックから秒を読み取る。}
```

```
32:
33: var
34:   Regs : Registers;
35:
36: begin
37:   with Regs do
38:     begin
39:       AX := $2C00;
40:       MsDos(Regs);
41:       ReadDosSecs := Hi(DX);
42:     end;
43: end;
44: {--------------------------------------------------------------}
45: begin{TimeRead body}
46:   if not(TimeLim in [1..59]) then
47:     begin
48:       ReadLn(S);
49:       Exit;
50:     end;
51:
52:     Now := ReadDosSecs;
53:     TimeUp := Now + TimeLim;
54:
55:     if TimeUp >= 60 then
56:       TimeUP := TimeUp -60;
57:     S := '';
58:     Getout := FALSE;
59:
60:     repeat
61:     { キーボード (コンソール) からの入力をチェックし、入力された文字は
62:       AL レジスタに格納される。}
63:
64:       Regs.AX := $0600;
65:       Regs.DX := Lo($0FF);
66:       MsDos(Regs);
67:
68:       ch := Chr(Lo(Regs.AX));
69:       case ch of
70:         NUL :  { 時間内に入力がない場合}
71:                    if ReadDosSecs = TimeUp then
72:                    begin
73:                        s := NUL;
74:                        GetOut := TRUE;
75:                    end;
76:
77:         CR :             { 入力はあったがリターンキーのみが押された) }
78:                    GetOut := TRUE;
79:
80:         eLse           { 有効な文字入力があった。入力文字を連結する}
```

```
81:         begin
82:           Write(ch);
83:           if Length(s) < (SizeOf(s) -1) then
84:              s := s + ch;
85:         end;
86:       end;
87:
88:    untiL GetOut;
89:    WriteLn;
90: end;
91: {No InitiaLization}
92: end.
```

1行目でこのユニットの名称をLIST4_4Aとしています。したがって、このユニットをディスク上にコンパイルした結果はLIST4_4A.TPUとして作られます。それはそれでかまわないのですが、本来はそのユニットが何だったのかがわかるような名称にすべきです。ここの例ではTREADなどとすべきでしょう。そうでないと、あとでこのユニットを他に流用しようとすると、はて何だったのだろうかとわからなくなります。

このユニットは7行目の手続きTimeReadだけしかありません。引数はキーボードから入力された文字列Sと時間制限秒数TimeLimです。制限秒数は1～59秒です。それ以外の制限秒数をTimeLimに指定すると、手続きTimeReadは単なる標準関数ReadLnと同じ動作しかしません。

プログラムの記述は12行目からです。12行目で後でMS-DOSのファンクションコールを使用しますからDosユニットの使用宣言をしておきます。

14行目から手続きTimeReadの実際の記述が始まります。20、21行目でおかしな^Mと^@がでてきます。これはアスキーコードで改行を表す13がCtrl-Mであること、またヌル文字を表す00がCtrl-@であることからきています。TURBO PascalではCtrl文字をそのまま定数としてプログラム内部で使えるのです。それぞれのCtrl文字が確かに13と00になっていることは先ほど開発したリスト4-2で確かめることができます。

23～28行目の変数定義の中にあるGetOutはこの制限時間内になにか入力があったか、それとも何もなくて制限時間が経過してしまったときにTRUEになります。つまり、この関数から外に出て行くべきかどうかを表す論理変数です。

30行目の関数ReadDosSecsはMS-DOSのファンクション・コールを利用して、呼ばれたときの時刻を読み出し時分秒のうち秒だけを正数値として返します。この関数はinterface部で宣言されていません。したがって、このユニットの外部から参照することはできません。このユニットをuses文で使う別プログラムは、あくまでもinterface部で公開されている手続きTimeReadしか使うことができないわけです。つまり、手続きTimeReadはパブリック、関数ReadDosSecsはプライベートなのです。

39行目でAレジスタに時刻読み出しのセットをして、40行目でファンクション・コー

ルをします。その結果としてDレジスタの上位1バイトに返された時刻の秒をこの関数値とします。

手続きTimeReadの実質部が45行目から始まります。46行目で制限秒数TimeLimが1～59の間にあるかを調べ、そうでなければ47～50行目で単なるReadLnで読み込みだけを行い、Exitによりこの手続きから外に出ます。46行目のif文はthen節しか持たず、else節がありません。それはExitにより外に出るようにしているからです。Exitを使わないと52行目以下をelse節にしなければなりません。その理由はみなさんでちょっと考えてみてください。

52行目で現在時刻の秒値を取り出し、53行目で制限秒数をそれに加えて制限時刻の秒値を作ります。その値TimeUpが60を越えているときには55、6行目で60を引いておきます。ちょっと見ると、今12時34分56秒だとすると、そこから20秒入力を待つとするとTimeLimが20ですから、TimeUpは16になります。そうなると20秒ではなく16秒しか待ってくれなくなるような気がします。

ところが、ここではReadDosSecsで時刻の秒値しか読みませんから、20秒後の12時35分16秒の時の秒値16がTimeUpでよいのです。これはTimeLimを1～59に制限しているおかげです。むろんちょっと改良すれば停電さえなければ何分後、いや何日後まで入力を待ち続けるプログラムを書くこともできます。しかし、一般的には実質的に1分間である59秒間あればよいでしょう。試しに画面を前にして1分間待ってみてください。1分間がかなり長い時間であることがわかると思います。

57、58行目で入力文字を空にして、この手続きから外へ出るのを制御する論理変数GetOutを初期化しておきます。

キー入力があるか制限時間が経過してしてしまうまで60～88行目を繰り返します。64～68行目でMS-DOSのファンクション・コールによりキーボードからの入力をチェックし、入力つまりキーの押下があったらその文字を変数chに取り込みます。

69～85行目のcase文ではファンクション・コールで得られた文字によりそれに応じた処理をします。70行目はchが空、つまりNUL、Ctrl-@である#00の場合です。これはキー押下がないときです。71行目で制限秒数になったかどうかを知らせれば、なっていたら入力文字sとして空を返します。それまでになにを入力していても制限秒数が経過してしまうとsは空になります。

77行目はchが改行(CR)の場合です。押下キーが改行であったということは入力が終了したことを意味しますから、GetOutをTRUEにして繰り返しから外に出ます。

80行目のelse節は改行キー以外のキー押下があった場合です。82行目でそのキー文字を表示し、83行目でsの最大長127文字を超えない限りchを次々にsにつけ加えます。

このユニットで定義する手続きTimeReadは90行目で終わりです。この後ろから最後のend部まではユニットが他のプログラムに結び付けられたときに最初に初期化し

ておくべきことがあればそれを書きます。ここでは必要ありません。

このユニットを TURBO Pascal でコンパイルします。そのときに Compile オプションを memory ではなく disk にしておかなければなりません。コンパイルが終了したらカレントディレクトリ上に LIST4_4A.TPU が作られていることを確認してください。もう一つ注意があります。それはユニット名、識別名は program の後ろに書く識別名同様に Pascal の規則に従うということです。たとえば、LIST4-4A のように間にマイナス記号を入れたりしてはいけません。しかし、そのソースファイル名として LIST4-4A.PAS のようにするのは MS-DOS の規則から許されます。

ところがこれをコンパイルすると LIST4-4A.TPU が作られます。そうなるとこのユニットを使用するプログラムは uses 文に LIST4-4A と書くことになります。これは TURBO Pascal でユニット識別名として許されませんから、そのプログラムのコンパイル時にエラーとなってしまいます。これを注意しないと TPU ファイルがディスク上に確かにあるにもかかわらず使用できないことになり、つまらないことが原因で仕事に手間取ります。

■リスト 4-4B

```
 1: program LIST4_4B;
 2:
 3: uses
 4:   LIST4_4A;
 5:
 6: const
 7:   TIMELIMIT = 30;{例として制限時間 30 秒}
 8:
 9: var
10:   answer : string;
11:
12: begin { main }
13:   WriteLn(TIMELIMIT,' 秒間で次の質問に答えて下さい。');
14:   Write('アメリカ合衆国の首都は？ ');
15:
16:   TimedRead( answer, TIMELIMIT);
17:
18:   if answer = 'Washington D.C.' then
19:     WriteLn('正解！　そのとうり!!')
20:
21:    eLse
22:     if answer = Chr(0) then
23:       WriteLn(^g^g,'時間切れです。!') {ブザーを警告}
24:
25:      eLse
26:       WriteLn('まちがいです！　答は"Washington D.C."です。');
27: end.
```

リスト 4-4B はこのユニットを使用してアメリカ合衆国の首都をあてさせるクイズを作った例です。7 行目で制限秒数として 30 秒を定数で設定しています。まず 13 、14 行目で問題を出します。16 行目で答え answer と制限時間 TIMELIMIT を引数としていま作ったユニットの手続き TimeRead を呼び出します。ここでは手続き TimeRead の素性(?)がよくわかっているからいいようなものですが、引数に定数を使うことはあまりやらないほうがよさそうです。呼び出した手続きの中で書き換えられてしまう危険性があるからです。そして定数だと思っていたら知らないうちに中身が変わっていたなどというデバッグで発見しにくい誤りが発生します。

18 ~ 26 行目は正解、誤答、時間切れの場合に応じた処理です。入力中に誤りを直そうとバックスペースするとそれも変数 answer の文字列として取り込まれてしまいますから、正解を入力しても 1 回で誤りなく入力しなければなりません。

そんなきびしい条件に加えて、Washington だけでは正解でなく D.C. を忘れるのは大目にみてほしいところですが、ここでは TimeRead のデモということでまあよいと思います。23 行目の ^g はブザー鳴動のコードです。これを画面表示すると文字が表示される代わりにブザーが鳴るように MS-DOS で取り決められているためです。

ここではまずユニットについて解説しました。ユニットを利用すれば interface だけを公開して、このユニットの中に入っている手続き関数はこんな機能だという説明を加えておけば、ソースコードではなく TPU ファイルだけで他のひとも使えるわけです。つまり、ソースコードを公開せずに便利なツールを商品化できます。

ソースコードを公開すると、プログラム開発のテクニック、ノウハウを知られてしまうので、ソフトハウスはソースコードを厳密に管理しています。したがって、こんなツールがあれば TURBO Pascal でプログラム開発が楽になるのにというツールを作っても公開・発売できなかったのです。ところがユニット機能により TPU ファイルとして商品化すれば、どうやってある手続きや関数をつくっているのかを知られることがなくなります。

さて、最後にここで開発した手続き TimeRead ですが、これは前にもいいましたが最終的なものと思わないでください。これを改良してもっとすぐれた手続きに変えることは可能です。または自分が開発するアプリケーション向けに改造するのもよいでしょう。たとえば、CAI(Computer Assisted Instruction：教育用ソフトウェア) で、練習問題に対する制限時間に使うなどの例が思い浮かびます。意外とおもしろい応用があるかもしれません。

ビジネスソフト開発の第１歩

5 ファイルを使う

TURBO Pascalでプログラムを作りながら、TURBO Pascalの強力なファイル関係の処理機能を使わずにいつも画面表示や印刷出力で結果を保存していませんか。

ファイル処理は慣れてしまえばとても便利なのですが、科学計算やグラフィックスの表示などにTURBO Pascalを使っていると、そちらの使用法を勉強するのが精いっぱいでファイルの勉強まで正直いって手が回りません。しかし、いつまでもそれではTURBO Pascalをフルに活用することはできません。そこで、この章ではファイル処理のトレーニングをやってみます。この章のプログラムを自分で作り、さらにそれを改造して動作を確かめれば、ファイル処理が案外簡単で便利なことが理解できるはずです。

この章では単なるトレーニング用のファイル処理ではなく実用的にも役に立つサンプル・プログラムを開発しています。初めにファイルの中身をのぞいてみるファイルダンプ、テキスト・レコードのソーティング、BASICファイルの変換を開発します。そして、ディスクの中身が変わっていないかどうかを調べるチェックとランダム・ファイルの取り扱いをトレーニングします。最後に標準のMS-DOSには入っていないフィアル表示用の外部コマンドを試作してみます。ここではある特定の日付以降作成・更新されたファイルを表示するコマンドをプログラミングしてみます。TURBO Pascalのマニュアルにはテキスト・ファイルの取り扱いなどの説明がありますが、この説明だけではよくわからないことが結構あります。また、出力装置、つまりプリンタか画面か、またはRS-232Cかなどの設定法も出力用のファイルとまったく同じに取り扱えることも例題がなければピンとこないでしょう。

まずは復習から

5-1■ファイルダンプ

ファイルの内容をそのまま画面表示するプログラムはMS-DOSのユーティリティにもありますし、そのほかにも多少プログラム開発をする人ならファイルダンプのソフトはPDSなどから手に入れて持っているでしょう。しかし、そのプログラムは一体どうやってファイルの中身を読みだしているのかを知っている人は少ないと思います。確かに、ファイルをオープンして１レコードずつ読んでくればよいのには違いありません

が、具体的なTURBO Pascalのプログラムとしてはどうやればよいのでしょうか。

リスト5-1がファイルダンプをTURBO Pascalでやるプログラムです。

10行目の定数MAX_BYTEは読み込んだファイルの内容を保持しておくためのバッファの大きさです。バッファとは入出力データを一時的に記憶しておく場所です。その大きさは0〜8191までですから、256H(＝8192)個、つまり、128バイト単位のレコードで64個分になります。

■リスト5-1

```
 1: program LIST5_1;
 2:
 3: Uses Crt;
 4:
 5: type
 6:   Str14 = string[14];
 7:
 8: const
 9:   BYTE_32  = 32;
10:   MAX_BYTE = 8191;
11:
12: var
13:   in_fiLe      :fiLe;       {入力ファイル}
14:   ch           : char ;     {PまたはC}
15:   out_dev      : text ;     {出力ファイル（装置）}
16:   fiLename     : Str14 ;    {入力ファイル名}
17:   fiLe_image   : array[0..MAX_BYTE] of byte;{ファイル内容}
18:   totaL_screen,             {総画面数}
19:   k,                        {画面カウンタ}
20:   disp_num,                 {表示カウンタ}
21:   disp_rec,                 {既表示画面数}
22:   Left_rec    : integer ;{未表示画面数}
23:
24: {***************************************************}
25:
26:   procedure Buffer_FLush ;{ファイル・イメージ初期化}
27:     var
28:       counter : integer ;
29:
30:     begin   {すべて0とする}
31:       for counter := 0 to MAX_BYTE do
32:         fiLe_image[ counter ] := 0 ;
33:     end; { procedure Buffer_FLush }
34:
35: {***************************************************}
36:
37:   function Found(fiLename:Str14):booLean;{指定ファイル存在チェック}
38:     var
39:       infiLe : fiLe ;
```

```
40:
41:     begin
42:       Assign(infiLe,fiLename);
43:         {$i-}
44:       Reset(infiLe); {ファイル-オープン}
45:         {$i+}
46:
47:       Found := (ioresuLt=0); {存在すれば TRUE}
48:     end;  { function Found }
49:
50: {*****************************************************}
51:
52:   procedure Hex_Disp(n:integer);{nの16進表示}
53:     var
54:       a,b   :integer;    {16進数各桁}
55:       chs   :string[16];{16進表示用文字列}
56:
57:     begin
58:       chs := '0123456789ABCDEF';
59:       a := Trunc(n/16);
60:       b := n - a*16;
61:
62:       if (a >=0) and (a <=15) then  {上位バイト}
63:         Write(out_dev,chs[a+1])
64:       eLse
65:         Write(out_dev,'?');
66:
67:       if (b >=0) and (b <=15) then  {下位バイト}
68:         Write(out_dev,chs[b+1])
69:       eLse
70:         Write(out_dev,'?');
71:
72:       Write(out_dev,' ');
73:     end; { procedure Hex_Disp }
74:
75: {*****************************************************}
76:
77:   procedure Dump_Disp;{画面表示}
78:     var
79:       i,j,L,n,m,            {カウンタ}
80:       offset      :integer;{表示先頭バイト数}
81:
82:     begin
83:       ClrScr; {見出しの表示}
84:       WriteLn(out_dev,' ':15,'FiLe Name : ',fiLename,
85:                 '         Part : ',
86:               k+1+disp_num*(BYTE_32 div 2),
87:               ' of ',totaL_screen);
88:       WriteLn(out_dev);
```

```
 89:    WriteLn(out_dev,'Offset',' ':24,
 90:                   'HexadecimaL',' ':25,'ASCII');
 91:    WriteLn(out_dev,'------  ',
 92:      '  -0--1--2--3--4--5--6--7--8--9--A--B--C',
 93:      '--D--E--F-   0123456789ABCDEF');
 94:    WriteLn(out_dev);
 95:
 96:    for i := 0 to 15 do
 97:      begin
 98:        L := k*256+i*16;
 99:        offset := disp_num*256+L;
100:        Write(out_dev,L:5,'    ');
101:            {16 進表示}
102:
103:        for j := 0 to 15 do
104:          begin
105:            n := offset+j;
106:            Hex_Disp(Ord(fiLe_image[n]));
107:          end; { for j }
108:
109:        Write(out_dev,' ');
110:            {アスキー表示}
111:        for j := 0 to 15 do
112:          begin
113:            n := offset+j;
114:            m := Ord(fiLe_image[n]);
115:
116:            if m > 127 then
117:              m := m-128;
118:
119:            if (m > 31) and (m <127) then
120:              Write(out_dev,chr(m)){アスキー文字}
121:            eLse
122:              Write(out_dev,' '); {非アスキー文字}
123:          end; { for j }
124:
125:        WriteLn(out_dev);
126:      end; { for i }
127:  end; { procedure Dump_Disp }
128:
129: {**********************************************}
130:
131: begin { main }
132:   Buffer_FLush; {初期化}
133:   ClrScr;
134:
135:   repeat{ファイル名入力}
136:     Write('  FiLe to Hex Dump -> '); ReadLn(fiLename);
137:   untiL Found(fiLename) or (Length(fiLename)=0);
```

```
138:
139:     if Length(fiLename)=0 then HaLt; {ファイルが空}
140:
141:     Assign(in_fiLe,fiLename);
142:     Reset(in_fiLe);
143:
144:     Write('   Output to (P)rinter/(C)onsoLe -> ');
145:
146:     repeat
147:       ReadLn(ch){出力装置指定}
148:     untiL UpCase(ch) in ['P','C'];
149:
150:     if UpCase(ch) = 'C' then
151:       Assign(out_dev,'CON')
152:     eLse
153:       Assign(out_dev,'PRN');
154:
155:     rewrite(out_dev);
156:
157:     Left_rec := fiLesize(in_fiLe);
158:
159:     if odd(Left_rec) then{総表示画面数}
160:       totaL_screen := (Left_rec div 2) +1
161:     eLse
162:       totaL_screen := Left_rec div 2;
163:
164:     disp_num := 0;
165:
166:     whiLe Left_rec > 0 do{表示開始}
167:       begin
168:         if BYTE_32 <= Left_rec then
169:           disp_rec := BYTE_32
170:         eLse
171:           disp_rec := Left_rec; {ファイル末尾}
172:
173:         BlockRead(in_fiLe,fiLe_image,disp_rec);
174:
175:         for k := 0 to ((disp_rec-1) div 2) do
176:           begin
177:             Dump_Disp;
178:             WriteLn;
179:             Write(' ':6,'<< Press <ESC> to Exit',
180:                         ' or Any Other Key to Continue >>');
181:             ch := ReadKey;
182:
183:             if Ord(ch)=27 then HaLt; {ESCキー押下}
184:           end; { for k }
185:
186:         disp_num := disp_num+1;
```

```
187:      Left_rec := Left_rec-disp_rec;
188:    end; { whiLe remainig }
189:
190:  WriteLn(out_dev);
191:  WriteLn(out_dev,'End of Process');
192:  end. { main }
```

■図 5-1

図 5-1 がリスト 5-1 を表示させたときの画面の様子です。

26 ~ 33 行目の Buffer_Flush 出続きは配列としてとった 256H 個のバッファ、file_image を 0 に初期化します。

37 ~ 48 行目の Found 手続きは引数 filename で指定されたファイルが存在するかどうかを調べるための関数です。その名前のファイルが存在すれば TRUE、しなければ FALSE を返します。この関数は存在しないファイルのダンプを取ろうとしたときにそれを止めさせるために必要です。メインプログラムの 137 行目で存在するファイル名かどうかを判定するときに使います。43 行目で {$i − } にして、再び 45 行目で {$i + } にするのは、存在しないファイル名を入力されたときに、44 行目の Reset 手続きが失敗して TURBO Pascal の I/O エラーで強制的にプログラムを終了されてしまうのを防止するための定石です。もしも filename に該当するファイルが存在しないときには ioresult にエラー番号が自動的に TURBO Pascal から入れられます。ioresult が 0 だとファイルが存在したことになります。

52 ~ 73 行目の Hex_Disp 手続きは 1 バイト、つまり 16 ビットで表される整数 n の上位と下位の 8 ビットを 16 進数で表示するためのものです。もしも、上位または下位が 15(FH) を越えるようなことがあったら、?マークを表示します。この手続きはとても利用価値があります。特にメモリの中身をのぞいてみるようなプログラムに表示用として流用できそうです。しかし、その目的には 00 ~ FF までの文字を関数の中で表示せず、そのまま引数として返すように改造したほうが使いやすいでしょう。

77 ~ 127 行目の少々長い手続き Dump_Disp は file_image の内容を表示します。このようにメインプログラムに入れてもいいようなプログラムでも手続きにすれば、メインプログラムの論理的構造がずっとすっきりします。94 行目まではダンプ結果を見やすくするための見出しの出力です。96 行目の i は表示の行、L は列です。disp_num は 256 個表示するかたまりとしてファイルを切ったときの何個目を表示するかを示す変数です。disp_num はメインプログラムの 164 行目で初期値が 0 にセットされています。103 行目から 16 進数でファイルの内容を表示し、111 行目からそれに対応するアスキーコードを表示します。

メインプログラムは 131 行目から始まります。まず始めにその内容を 16 進数でダンプしたいファイル名を入力するところです。もし、存在しないファイル名を入力すると 135 ~ 7 行目のループで何度も問い直してきます。[CR] だけ入力すると filename が空になりますから、139 行目で Halt により MS-DOS に戻ります。読み込んだ falename は in_file にして、ダンプ結果を表示するファイルを out_dev にします。この out_dev が CON ならコンソール (キーボードと CRT ディスプレイ)、PRN ならプリンタとまったくファイルと同じように取り扱えます。

リストの始めに戻って 13 行目を見ると、in_file は型なし (アンタイプト) ファイルとして定義されています。つまり、変数型として file のみで、それに続く of 以下が省略されています。この場合はディスクファイルの内部書式にかかわらずアクセスすることができます。ファイルは 128 バイト単位で区切って取り扱われます。一方、out_div のほうはテキストの出力、つまりテキストファイルですから変数型として text を指定します。

146 行で出力装置として C ならコンソール、P ならプリンタということになります。

148 行目の判定で入力された文字が大文字でも小文字でもよいように UpCase 関数を使っています。これは文字入力でなにかを選ばせるときにしょっちゅう使うテクニックです。

157 行目の Left_rec にはダンプすべきファイルの要素数が入ります。Left_rec は FileSize 関数の定義からは longint であるべきなのですが、ここではテスト用として小さなファイルのダンプしかやりませんので 22 行目で integer 型としています。1 要素は 128 バイトになります。一度に表示するバイト数は 256 バイトにしたので、159 ~ 162 行目で total_screen が表示すべき画面枚数になるわけです。

166 ～ 188 行は 32 × 128 ＝ 4096 バイト単位でファイルを読むための処理です。BlockRead は一度に読み込むバイト数が多いほど効率的になります。ファイルの最後のほうなどで 4096 バイトではファイルの終わりを越えてしまうというときには 171 行目で残りだけを読むようにしています。

175 ～ 184 行目は 4096 バイト読めたとき、つまり BYTE_32 が 32 のときには 4096 ÷ 16 ＝ 256 バイトを表示しています。256 バイトに足らずにファイルが終ってしまったときには、のこりの部分には buffer を初期化したときの 0 が表示されます。

ファイルの素早いソートには

5–2■テキストレコードのソート

ファイル内のレコードのある項目に着目して並び換えをするソーティングはビジネス・アプリケーションに欠かせません。パソコンが普及する以前に大型のコンピュータしかなかった時代から、いかに高速にソートするかはいろいろな研究がなされていますし、ソーティングの方法論は専門の書籍に必ずいくつか紹介されています。

ここではソーティング自体が目的ではなく、ファイル処理をトレーニングすることが目的ですから、どのソーティングが優れているかなどについては解説しません。ここで取り扱うファイルはテキストファイルです。テキストファイルは普通のファイルとは異なり、文字が基本要素です。したがって、各レコードは［CR］/［LF］(改行と行送り) で切られており、ファイルの最後は^z で終ります。また、テキストファイルに対しては入出力を同時に行うことはできませんし、シーケンシャルな処理しかできないので、非常に制約が多いといえます。また、Seek、Flush、Filepos、Filesize なども使えません。

しかし、テキストファイルにはすばらしい利点があります。それはエディタで直接データを作成したり修正したりできることです。TURBO Pascal で書いたプログラムに対する入力用のファイルなどをちょっと修正するときなどこの性質はとても役に立ちます。また、エディタが使えることは出力ファイルを他の目的に使うため加工するのにも便利だということです。

さて、ここでファイルのソートをプログラミングするのですが、ひとつ問題があります。それはワープロ・ソフトで作った住所録などは 2 バイトで表される日本語を使っていることです。たとえば、でたらめな順序で並んでいる名簿をアイウエオ順に整理するとします。英語ならアルファベットですから、まず 1 バイト目の英字をアスキーコード順に並べます。ところが、私の名字の小林だと「こ」ではなく、「しょう」の読みのほうに入れられます。まして、「東」(あずま) さんと「東田」(ひがしだ) さんになると同じ東のところに入れられてしまいます。そこでヒラガナのアスキー文字を「読み」としてレコードの中に入れ、それをソートすることを考えます。それでも、「茶谷」(ちゃた

に)さんが「近田」(ちかだ)さんより前にきてしまいます。「ャ」のほうが「カ」よりもアスキーコードでは前にあるためです。結局、日本語をアイウエオ順にソートすることはひとすじなわではいかないようです。どうもアスキーコードを決めるときの方法がそこまで配慮していなかったが原因といえそうです。むろん、促音や拗音や、濁点、半濁点の処理も考えてソートプログラムを作ればよいのですが、それはここでのファイル処理のトレーニングという主題からそれてしまいます。

そこで、ここではどのようにテキストファイルを処理するのかの例を示すため、アスキー文字をストレートにソートするだけにしました。リスト5-2がプログラムです。プログラム中の変数の主なものの意味はリスト中の書き込みを参考にしてください。

■リスト 5-2

```
 1: program LIST5_2;
 2:
 3: Uses
 4:          Crt;
 5:
 6: const
 7:  LINE_FEED       : char = #$0a;{ラインフィードコード}
 8:
 9: var
10:  input_Line,{キー入力}
11:  in_fiLename,{ソートファイル名}
12:  out_fiLename : string[14];{ソート出力ファイル名}
13:  fieLd1,
14:  fieLd2       : string[12];{キー入力作業変数}
15:
16:  origin_fiLe,{ソートファイル}
17:  sorted_fiLe  : text;{ソート出力ファイル}
18:
19:  fiLe_image   : array[1..32700] of char;{ファイル内容}
20:  rec_num,
21:  rec_pos      : array[1..5000] of integer;{ソート用ポインタ}
22:  char_num,
23:  char_pos     : array[1..10] of integer;{ポインタ用作業変数}
24:
25:  ch1,ch2      : char;{ソート作業文字変数}
26:
27:  key_Length,{キー長}
28:  key_num,{キー個数}
29:  key_pos1,
30:  key_pos2,{キー位置}
31:  nth_key,{n番目入力キー}
32:  rec_totaL,{総レコード数}
33:  fLg,{ソートフラッグ}
34:  i,j,k,L,n,p : integer;{作業用カウンタ}
35:
```

```
36: begin
37:  CLrScr;
38:  WriteLn('   Text Sort Program');
39:  WriteLn('----------------------');
40:  repeat
41:   WriteLn;
42:   Write('Enter Input FiLe Name -->');
43:   ReadLn(input_Line);
44:   if Length(input_Line) = 0 then begin {入力なし}
45:    WriteLn; WriteLn('end sort');
46:    HaLt;
47:   end;
48:   for i := 1 to Length(input_Line)
49:      do input_Line[i] := UpCase(input_Line[i]);
50:
51:   if Pos('.',input_Line) = 0 then  {拡張子指定なし}
52:     input_Line := Concat(input_Line,'.txt');{デフォールト}
53:
54:   in_fiLename := input_Line;
55:   Assign(origin_fiLe,in_fiLename);
56:   {$i-} Reset(origin_fiLe) {$i+};{ソートファイルオープン}
57:
58:   i := IOResult;
59:   if i <> 0 then
60:    WriteLn('UnabLe to open ',in_fiLename,
61:                     ' -- Check speLLing.')
62:  untiL i = 0;{オープン成功まで}
63:
64:  i := 0;{初期化}
65:  k := 1;
66:  rec_totaL := 0;
67: {ファイル内容をメモリに写す}
68:  whiLe not EOF(origin_fiLe) do begin
69:   i:= Succ(i);
70:   Read(origin_fiLe,fiLe_image[i]);
71:
72:   if fiLe_image[i] = LINE_FEED then begin
73:    rec_totaL := Succ(rec_totaL);
74:    rec_num[rec_totaL] := rec_totaL;
75:    rec_pos[rec_totaL] := k;
76:    k := Succ(i);
77:   end;
78:  end;
79:
80:  CLose(origin_fiLe);{以下メモリ内のみで作業}
81:  WriteLn;
82:  WriteLn(in_fiLename,' contains ',rec_totaL,' records.');
83:  nth_key := 0;
84:  WriteLn;
```

```
 85:
 86:  repeat{ソートキー入力  field1/field2}
 87:   Write('sort Key ',nth_key+1,' -->');
 88:   ReadLn(input_Line);
 89:   L := Length(input_Line);
 90:
 91:   if L <> 0 then begin
 92:    nth_key := nth_key + 1;
 93:    i := Pos('/',input_Line);
 94:    fieLd1 := Copy(input_Line,1,i-1);
 95:    fieLd2 := Copy(input_Line,i+1,L-i);
 96:    VaL(fieLd1,char_num[nth_key],j);
 97:    VaL(fieLd2,char_pos[nth_key],j);
 98:    char_pos[nth_key] := Pred(char_pos[nth_key]);
 99:   end;
100:
101:  untiL (L = 0) or (nth_key = 10);
102:
103:  if nth_key = 0 then begin{キー入力なし、ソート中止}
104:   WriteLn; WriteLn('end sort.  No sort performed.');
105:   HaLt;
106:  end;
107: {ソート開始}
108:  p := rec_totaL;
109:  WriteLn;
110:  Write('sorting ･････');
111:
112:  whiLe (p div 2) <> 0 do begin
113:   Write('.');{}ソート進行表示
114:   p := p div 2;
115:   k := rec_totaL - p;
116:   j := 1;
117:  {ソート}
118:   repeat
119:    i :=  j;
120:
121:    repeat
122:     L := i + p;
123:     n := 0;
124:     fLg := 0;
125:
126:     repeat
127:      n := Succ(n);
128:      key_pos1 := rec_pos[rec_num[i]] + char_pos[n];
129:      key_pos2 := rec_pos[rec_num[L]] + char_pos[n];
130:      key_Length := Abs(char_num[n])-1;
131:      key_num := -1;
132:
133:      repeat
```

```
134:     key_num := Succ(key_num);
135:     ch1 := fiLe_image[key_pos1+key_num];
136:     ch2 := fiLe_image[key_pos2+key_num];
137:
138:     if char_num[n] > 0
139:      then begin
140:       if ch1 > ch2 then fLg := 1;
141:       if ch1 < ch2 then fLg := 2; end
142:      eLse begin
143:       if ch2 > ch1 then fLg := 1;
144:       if ch2 < ch1 then fLg := 2; end;
145:
146:    untiL (fLg <> 0) or (key_num = key_Length);
147:
148:   untiL (fLg <> 0) or (n = nth_key);
149:
150:   if (fLg = 0) and (rec_num[i] > rec_num[L])
151:          then fLg := 1;
152:
153:   if fLg = 1 then begin
154:    n := rec_num[i];
155:    rec_num[i] := rec_num[L];
156:    rec_num[L] := n;
157:    i := i - p;
158:   end;
159:
160:  untiL (fLg <> 1) or (i < 1);
161:  j := j + 1;
162:
163: untiL j > k;
164: end;
165: {ソート終了}
166: WriteLn;
167:
168: repeat{出力ファイル名指定}
169:  WriteLn;
170:  Write('Enter Output FiLe Name -->');
171:  ReadLn(input_Line);
172:
173:  if Length(input_Line) = 0 then begin
174:   WriteLn; WriteLn('END SORT. ',
175:          'No output fiLe written.');
176:   HaLt;
177:  end;
178:
179:  for i := 1 to Length(input_Line)
180:      do input_Line[i] := UpCase(input_Line[i]);
181:
182:  if Pos('.',input_Line) = 0{デフォールト}
```

```
183:        then input_Line := Concat(input_Line,'.txt');
184:
185:     out_fiLename := input_Line;
186:     Assign(sorted_fiLe,out_fiLename);
187:     {$i-} Rewrite(sorted_fiLe) {$i+};
188:      i := IOResult;
189:
190:      if i <> 0 then WriteLn('unabLe to create ',
191:                       out_fiLename,' -- Try again.');
192:    untiL i = 0;
193:
194:    for k := 1 to rec_totaL do begin
195:     i := rec_pos[rec_num[k]] - 1;
196:
197:     repeat{ソート結果出力}
198:      i := Succ(i);
199:      Write(sorted_fiLe,fiLe_image[i]);
200:     untiL fiLe_image[i] = LINE_FEED;
201:
202:    end;
203:
204:    CLose(sorted_fiLe);
205:    WriteLn;
206:    WriteLn(out_fiLename,' contains ',
207:                          rec_totaL,' records.');
208:    WriteLn; WriteLn('END SORT');
209:   end.
```

40 〜 62 行まではこれからのソートの対象となるファイル名の入力です。input_Line という文字変数にソート用ファイルを入力しますが、何も入力せずに [CR] をタイプすると 44 〜 47 行目のところでプログラムの実行を停止します。49 行目では入力が小文字でなされたときには、ファイル名を大文字に変換します。51 行目は入力されたソート用のファイル名が拡張子を指定されていないときに、自動的に.txt の拡張子を付加するものです。たとえばこのリスト自体をソートの対象とするのであれば、LST5-1.PAS と入力します。40 行目の repeat に対する until は 62 行目にあります。ここまでの処理が成功しても、入力されたファイル名が存在しないと 56 行目の Reset で I/O エラーが出るので ioresult が 0 にならず正しいファイル名が入れられるまで繰り返します。存在するテキストファイル名が正しく入力されると、そのファイルを origin_file として 56 行目でオープンするのです。

68 〜 78 行目で origin_file 中のテキストレコードの文字を改行と行送りも含め配列 file_image にしまいます。テキストファイルはその要素に自由にアクセスできるようにはなっていないので、ソートするためには要素をメモリ中に入れて自由に取り出せるようにしておく必要があります。配列 rec_num はレコード番号のポインタで、ソートは

このポインタについて行います。この段階では rec_num には 1 から総レコード数が順番に入るだけです。72 行目でレコードの区切りである [LF] を検出します。配列 rec_pos には file_image に格納した各レコードの文字がどこから始まるかが入ります。こうしておかないと、文字がベタに file_image 入っているので、どこが各レコード、つまりテキストファイルを構成している要素の先頭なのかがわからなくなってしまうからです。

こうしてテキストファイルのすべての情報をこれらの配列に取り終ったら、つまりエンドオブファイル・マークを見つけたら、80 行目で入力ファイルをクローズします。82 行目で入力ファイルのレコード数を表示します。

ここまでで全体の 1/3 です。まだこの先がだいぶあるので嫌になってくるでしょう。それはどうなっているかを頭の中だけで考えていると、かなり精神的にストレスが生じるからなのです。それを避ける一番よい方法は小さな例を作って具体的にこのプログラムの流れを追うことです。そこで、ここでは簡単な例として、3つのレコードを持つテキストファイルを例にしてみましょう。次のようなレコードを持つテキストファイルが入力ファイルだったとします。

番号	内容
1	BBBBBB
2	CCC
3	AAAAA

このようなテキストファイルだと上で出てきた配列は、

```
file_image：BBBBBB[CR][LF]CCC[CR][LF]AAAAA
rec_num   ：1 2 3
rec_pos   ：1 9 14
```

のようになります。これを例にとりながらあとの処理を追いかけてみましょう。

86 ~ 101 行目でソートキーを入力します。入力の方法は a/b のようにします。この意味は a 個の文字列で b 番目から始まるものをソートするということです。ですから、始めの 2 個に注目してソートするなら 2/1 と入力します。したがって、a はテキストのレコードの最短のものの文字数を越えて入力するとエラーになります。一度のプログラム実行で最大 10 種類のソート指定が可能です。10 回ソートするのでなければ、途中でいつでも［CR］だけ入力すればソートキーの指定が終ります。この段階で、3/1 だけを指定したとすると、この例では、

```
char_num：3
char_pos：0
```

のそれぞれ 1 要素だけとなります。

98 行目で char_pos は入力値から 1 だけ少なくしていることに注意してください。103 行目でまったくソートキーを入力しなかったときを判定してメッセージを出し、プログラムの実行を停止します。

108～164行目がソーティングの本体です。110行目でソート中であることを知らせるメッセージを出します。メッセージの最後にピリオドがソート進行に伴って113行目により次々に表示されます。これによりこのプログラムのユーザーにプログラムが暴走して何もディスプレイに出ないのではなく、一生懸命(?)ソーティング中であることを知らせるのです。ここで用いているソート法はシェル・メッツナー法と呼ばれるものです。始めにもいったようにこの方法の詳細を説明することはしません。ただ、レコードそのものを操作するよりも、各レコードを指すポインタを操作するほうがはるかに高速にソートできることは憶えておいてください。何か大量のレコードを取り扱わなければならないような場合があったときには、非常に役立つテクニックです。このプログラムではファイルのコピーである file_image を入れ換えたりせずに、ただ各レコードの位置を示す rec_pos だけを入れ換えています。

170行目で出力すべきファイル名を聞いてきます。適当な名称のファイル名を入力します。ここでも拡張子を省略すると、自動的に.txt が付けられます。特にファイルとして取っておくのでなければ、コンソールである CON を指定します。そうすると、ソーティングの結果はディスプレイ画面に表示されます。もし、プリンタに印刷出力するのであれば、PRN を指定します。

このソートプログラムがいかに高速かは、このプログラム自体を入力ファイルとしていろいろなキーに対してソートさせてみればわかります。このソーティングの部分だけを取り出してライブラリにしておくと、ビジネス・アプリケーションの開発に便利です。

DOSさえ同じなら

5–3■BASICファイルの変換

BASIC を長年使っていて、これから TURBO Pascal に乗り換えようとしたときに必要な作業は、BASIC のソースコードを Pascal に書き換えることと、データファイルを書き直すことです。ソースコードの移植プログラムは私たちアマチュアには手が出せそうにもありませんが、データファイルの変換ならどうにかなりそうです。そこで、ここでは Microsoft-BASIC で作られたファイルを TURBO Pascal ファイルに変換します。

とはいっても、レコードがアスキー文字、整数、実数など色々種類がありますから、そのすべてに対応できるプログラムとなると簡単ではありません。ここで開発してみるプログラムは整数型の BASIC ファイルを TURBO Pascal ファイルに変換するものです。

Microsoft-BASIC では整数は2バイトずつアスキーコードで表されています。したがって、それを次々に読み込んで、それを TURBO Pascal の整数に変換してファイルを作ってやればよいはずです。

それでは早速プログラムを見てください。リスト 5-3 がそのプログラムです。32行目

まではこれから処理対象となるBASICファイル名、そして生成すべきTURBO Pascalファイル名の入力です。ここではファイル操作に対するエラー処理をまったくやっていません。もっと親切に作るのなら第3章でトレーニングした方法を使って、一応ディレクトリを表示して既存のファイルを破壊しないようにしてあげたほうがよいでしょう。

■リスト 5-3

```
 1: Program LIST5_3;
 2:
 3: uses Crt;
 4:
 5: Var
 6:   BASIC_name,{BASICファイル名}
 7:   PASCAL_name    :  string [14];{Pascalファイル名}
 8:   PASCAL_fiLe    :  fiLe of integer;{Pascalファイル}
 9:   BASIC_fiLe     :  text;{BASICファイル}
10:   record_image   :  array [1..500] of integer;{ファイル内容}
11:   a              :  char;{BASIC作業用文字}
12:   c,m,n,p,rec_num,
13:   fLg            :  integer;
14:
15: Begin
16:   CLrScr;
17:   WriteLn('まず、BASICのデータファイル名を',
18:                                         '入力してください');
19:   ReadLn(BASIC_name);
20:   WriteLn;
21:   WriteLn('次に、データの個数を',
22:                                         '入力してください');
23:   ReadLn(rec_num);
24:   WriteLn;
25:   WriteLn('最後に新しいPASCALのデータファイル名',
26:                              'を入力してリターンをどうぞ');
27:   ReadLn(PASCAL_name);
28:   Assign(BASIC_fiLe,BASIC_name);
29:   Reset(BASIC_fiLe);
30:   fLg  :=  0;
31:   n     :=  1;
32:
33:   for p := 1 to 500 do{ファイル内容初期化}
34:     begin
35:       record_image[p] := 0;
36:     end;
37:
38:   repeat{データ変換}
39:     Read(BASIC_fiLe,a);
40:     VaL(a,c,m);
41:     if m = 0 then
42:     begin
```

```
43:     if fLg < 2
44:       then
45:         begin
46:           record_image[n]
47:                   := (record_image [n] * 10) + c;
48:           fLg := 0;
49:         end
50:       eLse
51:         begin
52:           n := n + 1;
53:           record_image[n] := c;
54:           fLg := 0;
55:         end;
56:       end;
57:     fLg := fLg + 1;
58:   untiL eof(BASIC_fiLe);
59:
60:   CLose(BASIC_fiLe);
61:   Assign(PASCAL_fiLe,PASCAL_name);
62:   Rewrite(PASCAL_fiLe);
63:   WriteLn;
64:   WriteLn;
65:   WriteLn('変換されたデータは、次のとおりです');
66:   Write('| ');
67:   for n := 1 to rec_num do
68:     begin
69:       Write(record_image[n],' | ');
70:       Write(PASCAL_fiLe,record_image[n]);
71:     end;
72:   WriteLn;
73:   WriteLn;
74:   WriteLn;
75:   Reset(PASCAL_fiLe);
76:   WriteLn('新しいファイルのデータが、',
77:                 '次のとおりできました');
78:   Write('| ');
79:   for n := 1 to rec_num do
80:     begin
81:       Read(PASCAL_fiLe,record_image[n]);
82:       Write(record_image[n],' |');
83:     end;
84:   CLose(PASCAL_fiLe);
85: end.
```

28、29行目のAssignとResetはファイルをオープンするときの標準的な方法です。ここでは存在しないファイル名を入力したときの対策を省略してありますから、I/Oエラーが発生するとプログラムが停止してしまいます。けれども、このような変換プログ

ラムはBASICからTURBO Pascalへ乗り換えるときだけ一時的にに必要となるものですから、「凝った作り」にする必要はまったくないはずです。

30行ではまず読み初めですから、バイトポインタを0に設定し、変換結果を格納しておく配列record_imageのインデックス(添字)nを1個目にしておきます。33～36行目はその配列をすべて0にし、これで初期化が終ります。38～58行目がBASICの整数ファイルの読み込みとデータ変換です。ここで使用しているTURBO Pascalの標準手続きValが出てきます。

45行目のValでは、aとして読んだアスキー文字に対応する対応する数字データがないときにはmにエラーコードを帰します。m＝0のときはエラーがなかった、つまり読み込んだ文字aはBASICの整数データであったことがわかります。

BASICの整数データは2文字読み込まれ、始めに読み込まれたほうを10倍にし、後のほうをそのままにして加え合わせてrecord_imageに入れます。これで変換は終わりです。後はBASICファイルの終わりであるエンドオブファイル・マーク(略してEOF)を読むまでこれを続けます。60行目以下はrecord_imageを新しいTURBO Pascalファイルに書き込むだけです。72～84行はこうしてできたファイルの内容の画面表示です。

今日からこれなしにはいられない

5-4■チェックサム検査

最近はワープロの普及でフロッピーが日常生活の中によく使われるようになりました。確かにフロッピーに文書を入れておけば大量の「紙」を持ち歩く必要もありませんし、必要に応じて簡単に何度でも清書文書をつくることができます。このおかげで、スピーディにきれいな文章がだれにでも作れるようになったのです。ところが、フロッピーに情報をしまっておくことによる問題もあります。

たとえば、フロッピーをカバンの中に入れ、それとウォークマンのヘッドホンを一緒にしておくとフロッピーの内容が破壊されてしまいます。また、ワープロ入力の疲れ、ストレスに「効く」といわれている磁気ネックレスとか、エレキバンがフロッピーとハンドバッグの中で同居したりしても壊れてしまいます。ただし、壊れるといっても別にフロッピーが破れたりするような、外見上の形が変わるのではありません。フロッピーの内部で情報を記録している磁気パターンが乱されてパソコンやワープロに正しい情報を伝えられなくなるのです。

私の知っているひどい例では、フロッピーをなくさないようにとオフィスの同僚がマグネット画鋲でホワイトボードに貼っておいてくれたというのがありました。これほど極端な例はとにかく、このような「破壊」はフロッピーの外見からは診断不能です。それでもワープロのフロッピーなら文書を画面に表示してみれば、「破壊」の有無はすぐにわかります。ところがプログラムのように、中に何が入っているのかまったくわ

からないものは困ります。それがソースファイルであったとしても、ソースファイルのどこが壊されているのかを調べることは、プログラムが長いと大変に面倒な作業になります。さらに困ったことにはどれが誤りで、どれが正しいのかさえわからない場合もあります。

たとえば、磁気の影響で変数xがyに化け、しかもプログラムのなかにyという変数が他に使われていたりすると、私たちの目では絶対に見つかりません。おまけに、壊れている可能性のあるプログラムが私達には意味を持たないオブジェクト・ファイルだったり、数値データだったりしたらもうどうしようもありません。

このような磁気的記録をとっておく媒体の信頼性を向上させるために考えられたのが、ここで紹介するチェックサム・ベリファイ(検査)です。このテクニックは何も新しいものではなく、パソコン出現以前のコンピュータがガラス張りの電子計算室に入っていた時代の磁気テープに対して考案されました。当時は磁気テープや磁気テープ記録装置の信頼性が低かったのでこのような方法が必要だったのです。

しかし、現在でも前述のような「磁気嵐(?)」が日常的にあり、フロッピーがそれにさらされているのですから、このチェックサム・ベリファイは役に立ちます。先日、友人からプログラムを郵便で送ってもらったのですが、どこかで「何か」があったらしく、動くはずのプログラムが動かなくなっていました。そんなときにもフロッピーにチェックサムを付けておいてくれれば、友人が不完全なプログラムを送ったのか、フロッピーが「変質」したのかがはっきりするのです。

さらに、いまはやりのパソコン通信で友人とプログラムやデータをやり取りするときに心配なのは、本当に正しいプログラムが送受信できただろうかということです。モデムの信頼性や、電話回線のノイズのために本来の文字とは異なる文字が送受信される「文字化け」と呼ばれる現象がよくおこります。そんなときにもこのチェックサム・ベリファイは有効です。

それでは、チェックサム・ベリファイとはどんなものなのでしょうか。ここにフロッピーが１枚あるとしましょう。このフロッピーの中には何本かファイルがあります。ファイルはレコードの集合です。レコードはさらにバイト単位からできています。そのバイトは16ビットでできています。このビットはご存じのように０と１の値をとります。このファイルの中にあるビットの１の個数を勘定して、ファイルの情報としてフロッピーに付け加えておくのです。

こうしておけば、フロッピーの中のあるファイルが「賞味期限」を過ぎて変質しているかどうかは、ファイルをひとつずつ読み込んでその中に含まれる文字のビット１の数を数えて、それをさきほどの情報と比較して一致すればOK、そうでなければどちらかがおかしい、つまりファイルに何か変化が起こったことがわかります。

ここではファイルの中の各バイトを整数とみなして、その和をチェックサムとしました。１バイトは８ビットからできていますから、１ビットでも違えば整数としての値も

違ってくるはずです。

■リスト 5-4

```
 1: program LIST5_4;
 2:
 3: uses Crt,Dos;
 4:  type
 5:    TabLe_type = ^TabLe_Rec;{ファイル名リスト用ポインタ型}
 6:    TabLe_Rec  = record      {ファイル名リスト用型}
 7:                   disk_fiLename : string[12];{ファイル名}
 8:                   next_fiLename : TabLe_type;{次ファイルポインタ}
 9:                 end;
10:    Data_Rec   = record{ファイルデータ型}
11:                   fiLename_data : string[12];
12:                   checksum_data : integer;
13:                 end;
14:
15:  const
16:    VERIFY_DTA = 'VERIFY.DTA';{ベリファイ用ファイル名}
17:    BELL       = #7;{ブザー鳴動}
18:
19:  var
20:    Data_FiLe  : fiLe of Data_Rec;{ベリファイファイル}
21:    NameCheck  : Data_Rec;{ベリファイ対象ファイル}
22:    TabLe,
23:    ListPtr    : TabLe_Type;{リストポインタ}
24:    Drive_No   : string[2];{ドライブ番号}
25:    FiLe_Name  : string[12];{ファイル名}
26:    Check      : booLean;{ベリファイ}
27:    CheckSum   : integer;{チェックサム (バイト合計) }
28:    Character  : char;{ドライブ入力用}
29:    Path_Name  : String;{パス名}
30:    Attribute  : word;{アトリビュート}
31:    Files      : SearchRec;{Dos ユニット定義変数}
32:
33: {******************************************************}
34:
35:  procedure Disk_Check;{ベリファイファイル存在チェック}
36:
37:   var
38:     Data_FiLe : fiLe of char;
39:
40:   begin
41:    Assign(Data_FiLe,Concat(Drive_No,File_Name));
42:    {$I-}
43:    Reset(Data_FiLe);
44:    {$I+}
45:
46:    Check:=IOResuLt=0;
```

```
47:   if Check then
48:     CLose(Data_FiLe);
49:   end;
50:
51: {***************************************************}
52:
53:  procedure DirectLy_Check;
54:
55:   begin
56:     ListPtr:=niL;
57:     FindFirst('*.*',ANYFILE,Files);
58:     File_Name := Files.Name;
59:     if DosError<>18 then
60:       begin
61:        whiLe DosError<>18 do
62:          begin
63:            if((File_Name<>VERIFY_DTA){VERIFY.DTA 除外}
64:                and (File_Name<>'.'){ディレクトリ除外}
65:                and (File_Name<>'..'))
66:            then
67:              begin {ベリファイ対象ファイルのリスト作成}
68:                New(TabLe);
69:                TabLe^.disk_fiLename:=File_Name;
70:                TabLe^.next_fiLename:=ListPtr;
71:                ListPtr:=TabLe;
72:              end;
73:            FindNext(Files);
74:            File_Name := Files.Name;
75:          end;
76:
77:       end;
78:   end;
79:
80: {***************************************************}
81:
82:  procedure Read_Data;{チェックサムの作成}
83:
84:   var
85:     Check_FiLe       : fiLe;{チェックサム対象ファイル}
86:     Read_Byte        : array[1..50,1..128] of byte;{読み込みバイト}
87:     I,J,R_Size       : integer;
88:
89:   begin
90:     CheckSum:=0;
91:     Assign(Check_FiLe,Concat(Drive_No,File_Name));
92:     Reset(Check_FiLe);
93:
94:     repeat
95:       bLockread(Check_FiLe,Read_Byte,50,R_Size);
```

```
 96:      if R_Size>0
 97:      then
 98:        for I:=1 to R_Size do {読み込みバイトの総和}
 99:          for J:=1 to 128 do
100:            CheckSum:=(CheckSum+Read_Byte[I,J])
101:                           mod MAXINT;
102:    untiL R_Size=0;
103:
104:    CLose(Check_FiLe);
105:   end;
106:
107: {****************************************************}
108:
109:  procedure Write_Data;{VERIFY.DTA 作成}
110:
111:
112:   begin
113:    DirectLy_Check;
114:    if ListPtr=niL
115:    then
116:      WriteLn('コノ disk ニハ ナニモ ハイッテ イマセン。',BELL)
117:    eLse
118:      begin
119:        TabLe:=ListPtr;
120:        Assign(Data_FiLe,Concat(Drive_No,VERIFY_DTA));
121:        Rewrite(Data_FiLe); {VERIFY.DTA のオープン}
122:
123:        whiLe TabLe<>niL do{VERIFY.DTA の書き込み}
124:          begin
125:            File_Name:=TabLe^.disk_fiLename;
126:            Read_Data;
127:            NameCheck.fiLename_data:=File_Name;
128:            NameCheck.checksum_data:=CheckSum;
129:            Write(Data_FiLe,NameCheck);
130:            WriteLn(File_Name,' = ',CheckSum:6);
131:            TabLe:=TabLe^.next_fiLename;
132:          end;
133:
134:        CLose(Data_FiLe);
135:        WriteLn(VERIFY_DTA,' ヲ ツクリ マシタ。');
136:      end;
137:   end;
138:
139: {****************************************************}
140:
141:  procedure Verify_Data;{ベリファイ照合}
142:
143:   var
144:     Bad,Missing,
```

```
145:    Data_CheckSum   : integer;
146:
147:   begin
148:    DirectLy_Check;
149:    if ListPtr=niL
150:    then
151:      WriteLn('コノ disk ニハ ナニモ ハイッテ イマセン。',BELL)
152:     eLse
153:      begin{VERIFY.DTA のオープン}
154:       Assign(Data_FiLe,Concat(Drive_No,VERIFY_DTA));
155:       Reset(Data_FiLe);
156:       Bad:=0;
157:       Missing:=0;
158:
159:       repeat
160:        Read(Data_FiLe,NameCheck);
161:        File_Name:=NameCheck.fiLename_data;
162:        Data_CheckSum:=NameCheck.checksum_data;
163:        Disk_Check;
164:        if Check then
165:         begin
166:           Read_Data;
167:           if CheckSum=Data_CheckSum
168:           then{照合 OK}
169:             WriteLn('     *****  GOOD  ***** '
170:                                 ,File_Name)
171:           eLse{書き換えられている}
172:             begin
173:              WriteLn('     -----   BAD   ----- '
174:                                 ,File_Name,BELL);
175:              Bad:=Bad+1;
176:             end;
177:         end
178:        eLse{ファイルがない}
179:         begin
180:           WriteLn('     ===== MISSING ===== ',
181:                           File_Name,ELL);
182:           Missing:=Missing+1;
183:         end;
184:       untiL eof(Data_FiLe);
185:
186:       CLose(Data_FiLe);
187:       WriteLn('');{ベリファイ合計表示}
188:       WriteLn('');
189:       WriteLn('====================================',
190:                               '=============');
191:       WriteLn('');
192:       WriteLn('スベテノ fiLe ノ ベリファイ ガ オワリ マシタ。');
193:       WriteLn('コワレタ fiLe      : ',Bad:3);
```

```
194:      WriteLn('ショウメツ シタ fiLe  :',Missing:3);
195:      WriteLn('');
196:    end;
197:  end;
198:
199: {**************************************************}
200:
201: procedure Get_DriveNo;{ベリファイ対象ドライブ}
202:
203:  begin
204:   Write('ベリファイ スル ドライブ ヲ シテイ シテ クダサイ。');
205:   ReadLn(Character);
206:   Drive_No:=Concat(Character,':');
207:  end;
208:
209: {**************************************************}
210:
211: begin
212:  CLrScr;
213:  Get_DriveNo;{対象ドライブ入力}
214:  File_Name:=VERIFY_DTA;{ベリファイデータファイル名指定}
215:  Disk_Check;{ディスク存在チェック}
216:  if not Check
217:  then
218:    WriteLn(VERIFY_DTA,' ガ アリマセン。',BELL);
219:  WriteLn(Drive_No,' ドライブ ノ disk ニ ');{新たに VERIFY.DTA 作成}
220:  Write(' ',VERIFY_DTA,' ヲ アタラシク ツクリマスカ [Y to yes] ?');
221:  ReadLn(Character);
222:  if (Character='Y') or
223:  (Character='y') or
224:  (Character='ン')
225:  then
226:   Write_Data
227:  eLse
228:    begin
229:     if not Check
230:     then
231:       WriteLn(VERIFY_DTA,' ヲ ツクッテ クダサイ。',BELL)
232:     eLse
233:      begin
234:        Write('ベリファイ シマスカ [Y to yes ] ?');
235:        ReadLn(Character);
236:        if (Character='Y') or
237:        (Character='y') or
238:        (Character='ン')
239:        then
240:          Verify_Data{ベリファイ}
241:        eLse
242:          WriteLn('プログラム ヲ オワリ マス。',BELL);
```

```
243:       end;
244:     end;
245:  end.
```

リスト 5-4 を見てください。細かい内容をながめるよりもここでは各手続きの説明をしましょう。まず、35 行目の Disk_Check は File_Name で指定されたファイルが A：、B：、…などの Drive_No に存在するかどうかを調べます。その結果は論理変数 Check に格納されます。

53 ~ 78 行目の Directry_Check はチェックサム・ベリファイ用の情報を入れておくデータ・ファイル、VERIFY.DTA 以外のファイルがディレクトリ上に存在するかどうかを調べ、存在すればそれを次々にヒープ内に表として格納します。この作業はファイル用の手続きとして Dos ユニットに用意されている手続き、FindFirst と FindNext を使えば簡単にプログラミングできるはずです。詳しくは第２章を参考にしてください。

82 行目の手続き Read_Data は 50 × 128 バイト単位でファイル内のバイトを読み込み、128 バイトを 128 個の整数データとして和を次々に CheckSum として計算します。この CheckSum はどんどん大きくなり、ついには MAXINT(32767) を越えてしまい、システム・エラーが発生してプログラムの実行が停止します。それを避けるため 100 行目で 128 バイトの和を作るたびに MAXINT で割った余りを CheckSum にしています。

109 行目の手続き Write_Data は、先ほどヒープ上に確保したファイルの表からファイル名を読み出し、チェックサムとともに VERIFY.DTA に書き込むと同時に画面に表示します。141 行目の Verify_Data はこのプログラムの中心となる手続きです。

フロッピーに入っている VERIFY.DTA からファイル名とチェックサムを読んでは、フロッピーからヒープ上に移したファイルの表のデータと比較し、チェックサムが合わなければ Bad 、ファイル自体が存在しなければ Missing 、異常がなければ Good を表示します。201 行目の Get_DriveNo は読み込んだドライブ指定文字に：を付加するだけの簡単なものですから、説明の必要はないでしょう。

211 行目からメインプログラムです。まず、213 行目でベリファイ操作の対象となるフロッピーがどのドライブにセットされているのかを指定します。215 行目でそのドライブにセットされているフロッピー内にチェックサム情報を持っている VERIFY.DAT が入っているかどうかを調べ、入っていれば Check を TRUE 、そうでなければ FALSE にします。

FALSE の場合は、新たに VERIFY.DAT を作成するかどうかを決め、TRUE ならチェックサム・ベリファイを行います。私はあまり好きになれませんが、カナキーにセットして使う場合も考えて、「ン」を入力しても YES の Y(または y) を入力したのと同じ肯定入力にしています。さらに、入力を促すときにはピッと音を出していますが、耳ざわりなら定数 BELL の出力を除いてください。

こうして作ったプログラムを使って適当なディスクをチェックしてみた結果例をのせておきました。ここではいくつかのファイルが壊れていたり、消去されていた例を示します。

```
ﾍﾞﾘﾌｧｲ ｽﾙ ﾄﾞﾗｲﾌﾞ ｦ ｼﾃｲ ｼﾃ ｸﾀﾞｻｲ｡a
a: ﾄﾞﾗｲﾌﾞ ﾉ disk ﾆ
 VERIFY.DTA ｦ ｱﾀﾗｼｸ ﾂｸﾘﾏｽｶ [Y to yes] ?

ﾍﾞﾘﾌｧｲ ｼﾏｽｶ [Y to yes ] ?y
     *****  GOOD   ***** CONFIG.SYS
     *****  GOOD   ***** SAMPLE2.PAS
     -----   BAD   ----- SAMPLE3.EXE
     -----   BAD   ----- SAMPLE2.EXE
     *****  GOOD   ***** BINARY.DTA
     *****  GOOD   ***** COMMAND.COM
     ===== MISSING ===== LIST3-1.PAS
     ===== MISSING ===== CHECK.PAS


 ====================================================

ｽﾍﾞﾃﾉ fiLe ﾉ ﾍﾞﾘﾌｧｲ ｶﾞ ｵﾜﾘ ﾏｼﾀ｡
 ｺﾜﾚﾀ fiLe       :   2
 ｼｮｳﾒﾂ ｼﾀ fiLe  :   2
```

■図 5-2

アイデアひとつでスピードアップ

5-5■2進サーチ

かなり複雑なプログラムが続いたので、この節ではぐっとやさしいプログラムを作り一息入れましょう。5-2でテキストファイルの高速ソーティングをやりましたが、ファイル内のレコードをただソートしただけでは意味がありません。このファイルから必要なレコードを検索してこそソートした意味が生まれます。

そこで、ここではソート済みのファイルからいかに高速で検索するかを考えてみましょう。私たちが英語の辞書から目的の単語を探すときにどうするのかを考えてみましょう。たとえば、text という単語を引くときにどうするでしょうか。まず、辞書の「T」で始まる単語があるとおぼしきあたりを開きます。それがTで始まる単語のところでなく、Sで始まる単語のところだったらページを先にめくってTのところを出します。Tが出てきたらこんどはEのところを同じ用に探します。こうしてX、Tと探してtextがみつかります。

こんな当り前のことでもコンピュータにやらすとなると簡単ではありません。上の例のような方法をプログラムで作るには、検索の対象となるファイル自体をA、B、C、…のようにアルファベット見出しを付けて検索が容易となる構造に設計する必要があります。そのようなプログラムを作るのはそれほど難しいことではありませんが、単純な作業ほど効率よくこなせるというコンピュータの特徴を利用できる方法があります。ここで紹介する2進サーチがその方法です。

2進サーチというもっともらしい名前がついていますが、内容はそれほどたいしたものではありません。ファイルをまず半分に分け、検索すべきデータがどちらの半分に入っているのかをまず調べます。それが決まったら、こんどはその半分をさらに半分に分け、どちらの半分に入っているかを調べます。これを続けて行けばやがては検索すべきデータに当たるはずです。

こんな単純な方法ですが実に効果的です。頭から次々にレコードを読んでは目的のデータかどうかを判定するアルゴリズムで検索するのに比べると劇的な速度向上がえられます。むろん、この他にも速い方法はありますが、アルゴリズムが単純という点では2進サーチが優れているようです。

リスト5-5は2進サーチを動かしてみるためのテスト用データ・ファイルを作るプログラムです。べつにこんなプログラムがなくてもテスト・データは作れると思いますが、私も含めてものぐさな読者のために用意しました。2進サーチのプログラム本体はリスト5-6です。

■リスト5-5

```
 1: program LIST5_5;
 2:
 3: type
 4:   Str20 = string [20];
 5:
 6: var
 7:   DataFile  :file of Str20;
 8:   data      :array[1..16] of Str20;
 9:   i         :integer;
10:
11: begin
12:   data[1]  := 'AOYAMA';
13:   data[2]  := 'ENDO';
14:   data[3]  := 'Endo';
15:   data[4]  := 'ISHIHARA';
16:   data[5]  := 'Ishihara';
17:   data[6]  := 'KATO';
18:   data[7]  := 'KAWABATA';
19:   data[8]  := 'KOBAYASHI';
20:   data[9]  := 'KOGURE';
21:   data[10] := 'KURAMOTO';
```

```
22:   data[11] := 'MATUZAWA';
23:   data[12] := 'NAKAMURA';
24:   data[13] := 'OOTAKE';
25:   data[14] := 'SAKATANI';
26:   data[15] := 'SHIRAISHI';
27:   data[16] := 'TANAKA';
28:
29:   Assign(DataFile,'binary.dta');
30:   Rewrite(DataFile);
31:
32:   for i := 1 to 16 do
33:     Write(DataFile,data[i]);
34:   CLose(DataFile);
35: end.
```

■リスト5-6

```
 1: program LIST5_6;
 2:
 3: uses Crt;
 4:
 5: type
 6:   Str20 = string[20];
 7:
 8: var
 9:     datafiLe  :file of Str20;{検索対象ファイル}
10:     data,                    {読み込みデータ}
11:     sousaku   :Str20;        {検索データ}
12:     Lo,hi,                   {上、下、中央ポインタ}
13:     mid,
14:     count     :integer;      {検索回数}
15:     owari     :booLean;      {終了判定}
16:
17: begin
18:   CLrScr;
19:   Assign(datafiLe,'binary.dta');
20:   Reset(datafiLe);
21:   count := 0;
22:   Lo := 1;
23:   hi := FiLesize(datafiLe);
24:   owari := FALSE;{ここまで初期化}
25:   Write('検索する名前を入力してください。:');
26:   ReadLn(sousaku);{検索データ入力}
27:   repeat
28:     if Lo>hi
29:     then
30:       begin
31:         WriteLn('該当する名前はありません。');
```

```
32:       owari := TRUE;
33:     end
34:    eLse
35:     begin{検索開始}
36:       count := count+1;
37:       mid := (Lo+hi) div 2;{中央}
38:       WriteLn(count:2,' 回目, LO =',
39:         Lo:3,' HI =',hi:3,' MID =',mid:3);
40:       Reset(datafiLe);{データファイルオープンと読み出し}
41:       Seek(datafiLe,mid-1);
42:       Read(datafiLe,data);
43:       if data > sousaku then hi := mid-1;
44:       if data < sousaku then Lo := mid+1;
45:       if data = sousaku
46:       then
47:      begin
48:         WriteLn(mid:3,' 番目に入っています。');
49:         owari := TRUE;
50:      end;
51:     end;
52:   untiL owari;
53:   Close(datafiLe);
54: end.
```

プログラムはとても簡単ですから説明の必要もないでしょう。変数 Lo と hi は検索するデータレコード番号の下限と上限です。まず始めは 22 、3 行目でファイルの先頭と尻尾のレコード番号を Lo と hi とし、37 行目でそのまん中のレコード番号を mid とします。41 行目でファイルの先頭から mid − 1 個目までスキップし、42 行目で mid 番目のレコードを読み込みます。

そして、そのレコードが検索データより大きければ、検索データは mid と Lo の間にあったことがわかります。小さければ mid と hi の間にあります。mid と一致すれば検索は終わりです。43 ～ 45 行目がこの部分です。mid 番目のレコードは調べ済みですから、hi または Lo をそれぞれ mid − 1 と mid ＋ 1 に置き換えてまた同じことをします。

これを繰り返せばやがて目的のデータが mid 番目になるはずです。もし、目的のデータが存在しないと、最終的には hi と Lo の大小関係が逆になってしまいます。そのときには 28 行目の条件が成立し、32 行目で owari が TRUE になり 52 行目の until に当たって終了します。Halt を使ってプログラムから抜け出してもよいのですが、datafile をオープンしたままですので、TURBO Pascal がきちんとクローズしてくれるはずですがそれにたよらずオープンしたファイルをクローズして終了します。このプログラムで TANAKA を検索したときの出力結果を示しておきます。

この検索方法を見て、これなら再帰呼び出し (リカーシブ・コール) が使えるのではないかと思った読者はかなり Pascal が身に付いたひとです。ここではこれ以上 2 進サー

チプログラムの改良は行いませんが、自分で開発してこれと再帰呼び出しとどちらが高速か比較してみるとおもしろいでしょう。もっとも、その場合は乱数を使って、データ・ファイルが少なくとも1000個以上レコードを持つようにリスト5-5を改造しないと、比較にならないでしょう。

```
検索する名前を入力してください。:TANAKA
 1 回目, LO=  1 HI= 16 MID=  8
 2 回目, LO=  9 HI= 16 MID= 12
 3 回目, LO= 13 HI= 16 MID= 14
 4 回目, LO= 15 HI= 16 MID= 15
 5 回目, LO= 16 HI= 16 MID= 16
16 番目に入っています。

A>
```

■図 5-3

便利なファイル用外部コマンド

5–6■日付と属性でディレクトリ表示する

MS-DOSのコマンドにはファイル操作をするための基本コマンドがそろっていますが、それだけでは十分ではないようで、便利なコマンドがあちらこちらのパソコンネットにたくさんPDSとしてアップされています。市販のソフトよりもすばらしいPDSがいくつもあって、いつも感心してしまいます。しかし、PDSのほとんどは実行ファイルとして提供されるので、いったいどうやってそのプログラムの機能を実現しているのかはわかりません。

そんなところからどうも私はPDSを使う気にならないのです。コンピュータ・ウィルスのような悪意にもとづくものは当然として、善意のものでもプログラムの論理が誤っていてファイルを壊したりされては困ります。その点、ソースコードが公開されているプログラムなら中で何をやっているのかわかるので安心です。

ここではある特定の日付以降作成されたファイルや、今日から指定日数以前に作られたファイルを一覧にするプログラムを作ります。ファイルを次々に作っているとMS-DOSのdirコマンドではどれが目的のファイルかわかないほど大量のファイルが表示されてしまいます。あるファイルが特定の年月日以降に作られているはずということがはっきりしているときに、このプログラムは威力を発揮します。日付による制限でディレクトリ表示させるだけではおもしろくないので、ここではさらにファイルの属性も表示させることとしました。

リスト5-7を見てください。かなり長いプログラムなので全体の構成をつかむのが大変ですが、それにはメインプログラムから目を通すと早いでしょう。メインプログラムから見てゆくとして、その前に各種手続きの説明を簡単にしておきます。

まず38行目の手続きInstructはこのプログラムの使用法がよくわからなかったり、おかしな指定をしたときに使い方を説明するためのものです。

■リスト5-7

```
 1: program LIST5_7;
 2:
 3: uses Dos,Crt;
 4:
 5: const
 6:   MONTH_DAYS : array[ 0..12 ] of word =
 7:   ( 31, 31, 29, 31, 30, 31, 30, 31, 31, 30, 31, 30, 31);
 8:   {計算を簡単にするため0月は前年の12月にしておく}
 9:   SLASH      = '/';
10:   MAXLINE = 22;
11:   NULL       = ^@;
12:
13: type
14:   MaskForm      = string[ 80 ];
15:   FiLeRecPtr    = ^FiLeRecType;
16:   FiLeRecType  = record
17:                   FrName     : string[ 12 ];
18:                   FrDate     : string[ 20 ];
19:                   FrAttr     : string[ 15 ];
20:                   next       : FiLeRecPtr;
21:                end;
22: var
23:   FiLeList, {表示用のファイル情報}
24:   FirstList,
25:   EndList      : FiLeRecPtr;{並びの先頭と終り}
26:   FiLeMask      : MaskForm;
27:   FuLLDir       : booLean;
28:   back, {何日前か /P 指定}
29:   yr, mo, da,
30:   WeekDay         : word;
31:   DandT      : DateTime; { Dosユニット内で定義された型}
32:   PackedTime       : Longint;
33: FiLeTotaL,
34:   LineCount      : integer;
35:   PushKey         : Char;
36:   CmdLine     : string;  { 日付指定 /D }
37:
38: procedure Instruct;
39: begin
40:   WriteLn('このプログラムは2つの機能を持っています。');
```

```
41:   WriteLn;
42:   WriteLn('(1) 指定された日付以降に作られたファイルを表示する。');
43:   WriteLn('(2) 今日から指定された日数以前に作られたファイルを表示する。');
44:   WriteLn;
45:   WriteLn('標準形式: LIST5-7  [ mask ] [ /Dyy-mm-ddまたは/Pnn ]  [ /F ]');
46:   WriteLn;
47:   WriteLn('mask:表示したいファイル・マスク');
48:   WriteLn('yy-mm-dd:年月日');
49:   WriteLn('nn28日以内のさかのぼる日数 ( 1 <= n <= 28 )');
50:   WriteLn('/F:ファイル情報不要指定');
51:   WriteLn('注意!! /の後ろに空白を入れないこと。');
52:   HaLt
53: end;
54:
55: {-----------------------------------------------------------}
56:
57: procedure FindDate( Pktemp        : string;
58:                    var yr,mo,da : word );
59: { /Pオプション時の日付の並び替え }
60: var
61:   previous, x  : integer;
62: begin
63:   DeLete( Pktemp, 1, 2 );        { 文字列から /P を取り除く}
64:   VaL( Pktemp, previous ,x );
65:   if ( x > 0 ) or ( previous > 28 ) or ( previous < 0 )then Instruct;
66:    if ( previous > da ) then
67:      begin     { 前月までさかのぼる }
68:        da := previous - da;
69:        mo := mo - 1;
70:        da := MONTH_DAYS[ mo ] - da + 1;
71:        if ( mo = 0 ) then
72:          begin  { 1月から12月まで }
73:            mo := 12;
74:            yr := yr - 1;
75:          end
76:      end
77:    eLse
78:      da := da - previous + 1;
79: end;
80:
81: {-----------------------------------------------------------}
82:
83: procedure FiLeInfoList( var FiLeList : FiLeRecPtr );
84: {新しいFiLeListの並び替え}
85: var
86:   Ptr1, Ptr2 : FiLeRecPtr;
87: begin
88:   Ptr2 := FirstList;
89:   Ptr1 := Ptr2^.next;
```

```
 90:   EndList^.FrName := FiLeList^.FrName;
 91:   whiLe ( Ptr1^.FrName < FiLeList^.FrName ) do
 92:     begin      { ポインタを進める }
 93:       Ptr2 := Ptr1;
 94:       Ptr1 := Ptr2^.next;
 95:     end;
 96:   { Ptr1 と Ptr2 の間に挿入 }
 97:
 98:   FiLeList^.next := Ptr1;
 99:   Ptr2^.next := FiLeList;
100: end;
101:
102: {-----------------------------------------------------------------}
103:
104: procedure FiLeInfoDisp( PkCompDate       : LongInt;{日付}
105:                     var FiLeInfoRec   : SearchRec );{ディレクトリ情報}
106: { ディレクトリ探索で見つかったファイル名を表示}
107: type
108:   DandTStr = string[ 20 ];
109: var
110:   PkFiLeDate  : Longint;{パックされたファイル日付}
111:   DateAndTime : DandTStr;{展開されたファイル日付}
112:
113: procedure TimeStamp( var PkTime : Longint;
114:                    var dStr   : DandTStr );
115: { 圧縮されたファイル日付を変換 }
116: var
117:   stamp       : DateTime;{ Dos ユニット内の型 }
118:   stampStr,
119:   workStr     : string[ 20 ];
120:
121: begin {タイムスタンプの展開と先頭の0付加}
122:   UnpackTime ( FiLeInfoRec.Time, stamp );
123:
124:   Str( stamp.Year - 1900 , stampStr );
125:   Str( stamp.Month, workStr );
126:   if ( stamp.Month < 10 ) then
127:     stampStr := stampStr + '-0' + workStr
128:   eLse
129:     stampStr := stampStr + '-' + workStr;
130:   str( stamp.Day, workStr );
131:   if ( stamp.Day < 10 ) then
132:     stampStr := stampStr + '-0' + workStr + '    '
133:   eLse
134:     stampStr := stampStr + '-' + workStr + '    ';
135:
136:   Str( stamp.Hour, workStr );
137:   if( stamp.Hour < 10 ) then
138:     stampStr := stampStr + '0' + workStr
```

```
139:    eLse
140:     stampStr := stampStr + workStr;
141:
142:   Str( stamp.Min, workStr );
143:   if ( stamp.Min < 10 ) then
144:     stampStr := stampStr + ':0'
145:    eLse
146:     stampStr := stampStr + ':';
147:   dStr := stampStr + workStr;
148: end;
149:
150: begin
151:   PkFiLeDate := FiLeInfoRec.Time;
152:   if ( PkFiLeDate >= PkCompDate ) then
153:     begin      { 指定日付かそれ以降のファイル }
154:       New( FiLeList );
155:       FiLLchar( FiLeList^, SizeOf(FiLeList^), NULL );
156:       timeStamp( PkFiLeDate, DateAndTime );
157:       inc( FiLeTotaL );
158:       FiLeList^.FrName := FiLeInfoRec.Name;
159:       if FuLLDir then
160:         begin
161:           FiLeList^.FrDate := DateAndTime;
162:           if ( FiLeInfoRec.Attr = 32 ) then
163:               { アーカイブビットがオン }
164:             FiLeList^.FrAttr := 'アーカイブ'
165:            eLse
166:             if ( FiLeInfoRec.Attr = 16 ) then
167:               FiLeList^.FrAttr := 'ディレクトリ'
168:              eLse
169:               if ( FiLeInfoRec.Attr = 8 ) then
170:                 FiLeList^.FrAttr := 'ボリュームラベル'
171:          eLse
172:            FiLeList^.FrAttr := 'その他'
173:         end;
174:       FiLeInfoList( FiLeList );  { FiLeList の挿入 }
175:     end;
176: end;
177:
178: {----------------------------------------------------------------------}
179:
180: procedure DirSearch( PkTime : Longint;  { 比較用の圧縮日付 }
181:                      mask   : MaskForm );  { ディレクトリ探索マスク }
182: { マスクでカレントディレクトリを探索する }
183: var
184:   FiLes : SearchRec;     { Dos ユニットの型 }
185: begin
186:   FiLeTotaL := 0;
187:   FindFirst( mask, AnyFiLe, FiLes );
```

```
188:   if ( DosError = 0 ) then
189:     repeat
190:       FiLeInfoDisp( PkTime, FiLes );     { 見つけたファイルの表示 }
191:       FindNext(FiLes)
192:     untiL (DosError <> 0 );         { 全ての適合ファイルを探す }
193: end;
194:
195: {-----------------------------------------------------------------}
196:
197: procedure CmdLineParam( var CmdLine :string ) ;
198: {コマンドラインのパラメタから日付を取り出す }
199: var
200:   j, k : integer;
201:   temp : string[ 15 ];
202: begin
203:   { CmdLine の中に日付が入っているので/D を取り除く }
204:   DeLete( CmdLine, 1, 2 );
205: { 年 }
206:   j := Pos( '-' ,CmdLine );
207:   temp := Copy ( CmdLine, 1, j-1 );
208:   DeLete ( CmdLine, 1, j );
209:   VaL( temp, yr, j);
210:   yr := yr + 1900;
211: { 月 }
212:   j := Pos( '-', CmdLine );
213:   temp := Copy( CmdLine, 1, j-1);
214:   DeLete ( CmdLine, 1, j );
215:   VaL( temp, mo, j);
216: { 日 }
217:   temp := Copy( CmdLine, 1, j-1);
218:   VaL( CmdLine, da, j);
219: end;
220:
221: {-----------------------------------------------------------------}
222:
223: procedure ParamAnaL;{コマンドパラメタ解析}
224: var
225:   j,k : integer;
226:   temp: string[ 15 ];
227: begin
228:   k:= ParamCount;
229:   for j := 1 to k do
230:     begin
231:      temp := ParamStr( j );
232:      if ( temp[ 1 ] = SLASH ) then
233:        case UpCase( temp[ 2 ] ) of
234:          'P' : FindDate( temp, yr, mo, da );
235:          'D' : begin
236:                  if Length ( temp ) > 2 then
```

```
237:                    begin
238:                      CmdLine := temp;
239:              CmdLineParam( CmdLine )
240:                    end
241:                   eLse
242:                    begin
243:                      yr := 1985;
244:                      mo := 1;
245:                      da := 1;
246:                    end
247:                 end;
248:       'F' : FuLLDir := FaLse;
249:       eLse {パラメタ誤り}
250:         begin
251:           WriteLn( 'オプションが誤っています');
252:           Instruct
253:         end
254:     end
255:    eLse {残った文字はファイルマスク}
256:     FiLeMask := temp
257:   end;
258: end;
259:
260: {------------------------------------------------------------------------}
261:
262: procedure ListDir(var FirstList, EndList : FiLeRecPtr );{画面表示}
263: var
264:   Ptr : FiLeRecPtr;
265: begin
266:   Ptr := FirstList^.next;
267:   whiLe ( Ptr <> EndList ) do
268:     begin
269:       WriteLn( Ptr^.FrName:15, Ptr^.FrDate:25, Ptr^.FrAttr:15 );
270:       Ptr := Ptr^.next;
271:       Inc( LineCount );
272:       if ( LineCount MOD MAXLINE = 0 ) then
273:         begin
274:           WriteLn( 'キーを押してください');
275:           PushKey := ReadKey
276:         end
277:     end;
278:   WriteLn( FiLeTotaL:8, ' 本のファイルがあります' );
279: end;   { ListDir }
280:
281: {------------------------------------------------------------------------}
282:
283: begin{メインプログラム}
284:   if (ParamStr( 1 ) = '?' ) then
285:     Instruct;
```

```
286:   New( FirstList );
287:   New( EndList );
288:   FirstList^.next := EndList;   { リストの初期化 }
289:   GetDate( yr, mo, da, WeekDay );       { デフォールトは今日の日付 }
290:   FiLemask := '*.*';   { 全ファイル }
291:   FuLLDir := True;         { 日付とアーカイブ状態を表示 }
292:   LineCount := 0;          { 表示行数 }
293:   if ( ParamCount = 0 ) then    { デフォールトの探索 }
294:     FiLeMask := '*.*'
295:   eLse ParamAnaL;   { コマンドライン解析 }
296:
297:   with DandT do
298:     begin
299:       Year  := yr;
300:       Month := mo;
301:       Day   := da;
302:       Hour  := 0;
303:       Min   := 0;
304:       Sec   := 0;
305:     end;
306:   PackTime( DandT, PackedTime );{今日の日付パック}
307:   DirSearch( PackedTime, FiLeMask );{ディレクトリ探索}
308:   ListDir( FirstList, EndList );{ディレクトリ表示}
309: end.
```

このプログラムをディスク上にコンパイルした結果はプログラムファイル名によりますが、ここではプログラムファイル名としてLIST5-7.PASとしたとしましょう。そうすると、ディスク上にはLIST5-7.EXEが生成されます。それを実際に走らせるには、たとえば、

```
A>LIST5-7
```

のようにします。ところが、実はこのプログラムは日付や日数やコマンドパラメータとして必要とするのです。

たとえば、1991年1月1日以降に作られたファイルの一覧を見たいときには、

```
LIST5-7 /D1911-1-1
```

と入力します。もしも、15日前から作られたファイルを見たければ、

```
LIST5-7 /P15
```

と入力します。

そしてコマンドパラメータを入れない場合にはプログラムの使用法を表示して停止するようにしています。つまり、このプログラムが一体何をするプログラムなのかを説明するようになっているのです。こうしておけば、コマンドパラメータの入れ方がわからなくなっても安心です。内容をよく覚えていないプログラムをなんど走らせてもうまく使えないことほどイライラするものはありません。あたながほかのひとにもぜひ使ってほしいと思うプログラムを作り、それをフリーウェアにするのならぜひこの機能をつけるべきです。なんだかわからないプログラムがひたすら、

```
Illegal command.  Execution aborted.
```

のような表示を出すとしたら腹が立つだけで使ってみる気はおこりません。

さて、38行目から始まるのがそのInstructです。内容はプログラムの使用法を画面表示するだけですから、むずかしいところは何もありません。この手続きが呼ばれるのは使用法がわからなかったり、パラメータ指定が誤っていたりという場合ですから画面表示が終わったら52行目でHaltによりプログラムの実行を終了します。

57行目の手続きFindDateは/Pにより何日前と指定されたときに、今日からその日数前の日付は何だったのかを決定します。63行目では文字列/Pは日数に関係ないのでを取り除きます。64行目で文字列として入力されている日数を数値データに変換します。ここで、Val手続きの3番目のパラメータxは数値データへの変換がうまくゆけば0、そうでないときにはxの値は0以外の未定義の値が入れられます。

65行目でそのxが0以外になっているか、日数が28を越えているか、あるいは負になっているかをチェックします。日数の最大値を28としたのは、2月が28日までということと、一応4週間前まであればよいということからこうなりました。

66行目では指定日数をさかのぼると前月になるかどうかを判定します。71行目で月を表す変数moが0かどうかの判定があります。0月というのは実際上はありませんが、ここではプログラムが簡単になるように前年の12月を計算の便宜上0月としています。

83行目の手続きFileInfoListはパラメータで指定された条件に一致するファイルの情報をポインタを使用したリスト構造につけ加えるためのものです。このテクニックはポインタ使用の基礎ですから、もうこのエキスパート・マニュアルを読んでいるあなたにはおなじみなはずです。

104行目の手続きFileInfoDispがこのプログラムの中心になります。この手続きの中には内部で使用する手続きTimeStampが113行目から定義されています。

117行目の変数stampはDateTime型に宣言されています。このDateTime型はDosユニット内で宣言されているレコード型で、

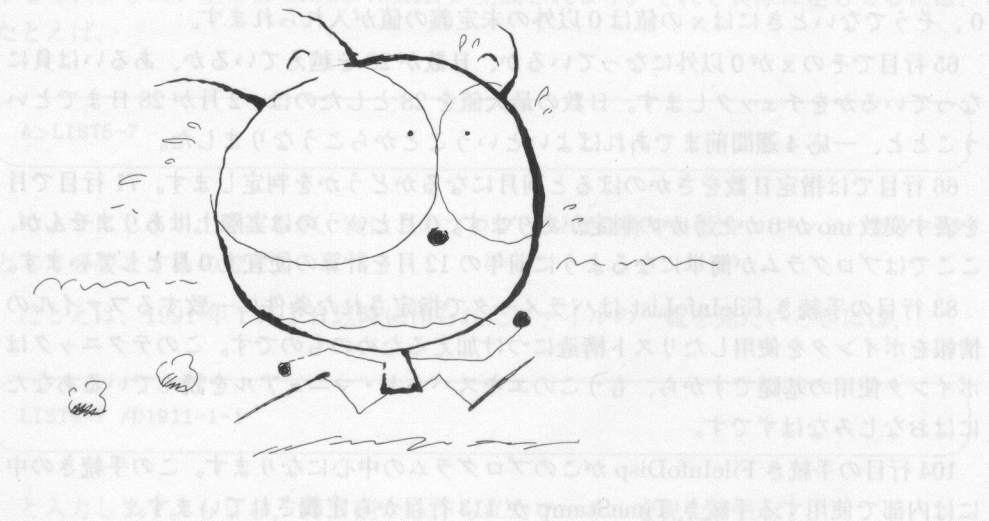
```
DateTime = record
             Year,Month,Day,Hour,Min,Sec : word
           end;
```

の形式です。この手続きによりFileInfoRecのフィールドTimeにlongint型でパックされている日付と時刻をstampに展開します。

124〜147行目は各フィールドが一桁だったときに先頭に0を付加するための処理です。150行目から指定日付に対して各ファイルのパックされた日付と時刻をlongint形式で比較して表示すべきファイルかどうかを調べています。

180行目の手続きDirSearchではマスク指定されたファイルを探索して、先ほどの手続きFileInfoDispで日付が適合するファイルを表示します。197行目のCmdLineParamはこのプログラムにコマンドラインパラメータとして/Dで文字列として日付が引き渡されたときにそれを取り出します。

223行目から始まる手続きParamAnalはパラメータが/D、/P、/Fなので、それに応じて日付を作り出しています。/Dで解析できないパラメータがあったときには1985年1月1日を日付としています。262行目の手続きListDirはリスト構造に作られた日付適合ファイルを画面表示します。

これで手続きの定義はすべてです。283行目から始まるメインプログラムでやっているのはこれら手続きを次々に使って処理をするだけです。プログラム中のコメントを読めば解説の必要はないでしょう。

6 ターボ・マウス

賢いねずみを飼い慣らせ

ひところキーボードに代わる入力装置としてマウスの人気が異常に高くなったことがありました。確かに、アップルのマッキントッシュに使用されたマウスは、Mac-OSのすばらしさともあいまって、非常に優れたユーザー・インタフェースを提供しました。しかし、国産パソコンとソフトにはいわゆる「お絵描き」ソフトを除いては本格的なマウス使用環境が整わず、いつのまにかマウスはパソコンの横でホコリをかぶる存在になってしまったようです。

しかし、このままではせっかく投資したお金が無駄になってしまい残念です。上手に使えば、マウスほど安価で効果的なポインティング・デバイスは他にありません。それにMS-Windows3.0のリリースに伴ってウィンドウ環境も整いつつあります。いよいよマウスを本格的に使う時代が到来したようです。そこで、ここではマウスをTURBO Pascalで活用するにはどうすればよいのかを解説します。

もっとも、本格的なマウス活用となると、マウスで何を選択するかを図形で画面に表示するアイコンの設計までやらなければなりません。アイコンとしてどんな図形がユーザーに直感的に理解できるかまで含めると、大きな研究テーマになってしまいます。

また、マウスを使って絵を描くこともまじめにコンピュータ・グラフィックスとして解説したのではこの本の範囲を越えてしまいます。マウスのグラフィックスへの利用は同じPUGシリーズに「ターボグラフィックス」という本がありますのでそちらを参考にしてください。この本もTURBO Pascalで書いてありますから、きっと役に立つと思います。

さて、ここで作る簡単なプログラムはマウス用のデバイス・ドライバであるmouse.sysの存在を前提にしています。むろんこれがなくてもユーザーズ・マニュアルにあるI/Oポートの情報をもとになんとかマウス動作をプログラミングすることは可能ですが、かなり面倒なことになります。マウスは市販のアプリケーション・ソフトでしか使わないということなら結構ですが、ひとつ本格的なマウス利用プログラムを開発してやろうと思うのなら、mouse.sysを手に入れておくことを是非お勧めします。

このデバイス・ドライバがあればほとんどのマウス動作をサポートすることが可能です。なにしろ、マウス操作をすべて割り込みで処理しているので、TURBO Pascalでのプログラミングがぐっと簡単になります。ただ、mouse.sysに付属してくる資料は

BASICが例になっていますし、読んでもすぐにわかるようにはできていません。

この章ではごく利用頻度の高い基本的ルーチンを紹介することとします。これを理解できれば、付属資料が何をいっているのかがよく理解できるでしょう。ここで使用しているのはASCIIまたはNECから提供されているMicrosoftマウスです。つまり、2ボタン方式のマウスです。これをサポートするのがmouse.sysです。

mouse.sysをあなたのパソコンに組み込むのは簡単です。システム立ち上げ用のフロッピーに入っているconfig.sysにdevice = mouse.sysを付け加えるだけです。

■リスト 6-1

```
 1: program LIST6_1;
 2: Uses Crt,Dos;
 3:
 4: const
 5:     MOUSE_INT = $33;
 6:
 7: type
 8:   CursorForm = array[0..31] of word;
 9:
10: {ファンクション 0}
11: function InitiaL : integer;
12: {マウス環境の初期化}
13:   var
14:     dos_reg : Registers;
15:
16:   begin
17:     dos_reg.ax := 0;
18:     Intr(MOUSE_INT,dos_reg);
19:     InitiaL := dos_reg.ax;    {環境状態}
20:   end;
21: {----------------------------------------------------------}
22: {ファンクション 1}
23: procedure Indication;
24: {カーソル表示}
25:   var
26:     dos_reg : Registers;
27:
28:   begin
29:     dos_reg.ax := 1;
30:     Intr(MOUSE_INT,dos_reg);
31:   end;
32: {----------------------------------------------------------}
33: {ファンクション 2}
34: procedure Erase;
35: {カーソル消去}
36:   var
37:     dos_reg : Registers;
38:
```

```
39:   begin
40:     dos_reg.ax := 2;
41:     Intr(MOUSE_INT,dos_reg);
42:   end;
43: {-----------------------------------------------------------}
44: {ファンクション3}
45: function Position (var positionX,
46:                        positionY : integer) : integer;
47: {カーソル位置の取得}
48:   var
49:     dos_reg : Registers;
50:
51:   begin
52:     dos_reg.ax := 3;
53:     Intr(MOUSE_INT,dos_reg);
54:     Position  := 10*dos_reg.ax
55:                + dos_reg.bx;{左右ボタン}
56:     positionX := dos_reg.cx;    {水平座標}
57:     positionY := dos_reg.dx;    {垂直座標}
58:   end;
59: {-----------------------------------------------------------}
60: {ファンクション4}
61: procedure Set_Position (positionX,positionY : integer);
62: {カーソル位置の設定}
63:   var
64:       dos_reg : Registers;
65:
66:   begin
67:     dos_reg.ax := 4;
68:     dos_reg.cx := positionX;   {水平座標}
69:     dos_reg.dx := positionY;   {垂直座標}
70:     Intr(MOUSE_INT,dos_reg)
71:   end;
72: {-----------------------------------------------------------}
73: {ファンクション5}
74: function Left_Press ( var count,
75:                           LastX,
76:                           LastY : integer) : integer;
77: {左ボタン押下情報の取得}
78:   var
79:     dos_reg : Registers;
80:
81:   begin
82:     dos_reg.ax := 5;
83:     Intr(MOUSE_INT,dos_reg);
84:     Left_Press := 10*dos_reg.ax;{押下情報}
85:     count    := dos_reg.bx;      {押下回数}
86:     LastX := dos_reg.cx;         {押下時水平座標}
87:     LastY := dos_reg.dx          {押下時垂直座標}
```

```
 88:  end;
 89: {--------------------------------------------------------}
 90: {ファンクション6}
 91: function Left_ReLease ( var count,
 92:                             LastX,
 93:                             LastY : integer) : integer;
 94: {左ボタン解放情報の取得}
 95:   var
 96:     dos_reg : Registers;
 97:
 98:   begin
 99:     dos_reg.ax := 6;
100:     Intr(MOUSE_INT,dos_reg);
101:     Left_ReLease := 10*dos_reg.ax;{押下情報}
102:     count := dos_reg.bx;          {解放回数}
103:     LastX := dos_reg.cx;          {解放時水平座標}
104:     LastY := dos_reg.dx;          {解放時垂直座標}
105:   end;
106: {--------------------------------------------------------}
107: {ファンクション7}
108: function Right_Press ( var count,
109:                            LastX,
110:                            LastY : integer) : integer;
111: {右ボタン押下情報の取得}
112:   var
113:     dos_reg : Registers;
114:
115:   begin
116:     dos_reg.ax := 7;
117:     Intr(MOUSE_INT,dos_reg);
118:     Right_Press := dos_reg.ax;{押下情報}
119:     count   := dos_reg.bx;    {押下回数}
120:     LastX := dos_reg.cx;      {押下時水平座標}
121:     LastY := dos_reg.dx       {押下時垂直座標}
122:   end;
123: {--------------------------------------------------------}
124: {ファンクション8}
125: function Right_ReLease ( var count,
126:                              LastX,
127:                              LastY : integer) : integer;
128: {右ボタン解放情報の取得}
129:   var
130:     dos_reg : Registers;
131:
132:   begin
133:     dos_reg.ax := 8;
134:     Intr(MOUSE_INT,dos_reg);
135:     Right_ReLease := dos_reg.ax;{押下情報}
136:     count := dos_reg.bx;        {解放回数}
```

```
137:     LastX := dos_reg.cx;           {解放時水平座標}
138:     LastY := dos_reg.dx;           {解放時垂直座標}
139:   end;
140: {--------------------------------------------------}
141: {ファンクション 9}
142: procedure Set_Shape (centerX,centerY : integer;
143:                                shape  : CursorForm);
144: {カーソル形状設定}
145:   var
146:     dos_reg : Registers;
147:
148:   begin
149:     dos_reg.ax := 9;
150:     dos_reg.bx := centerX;    {カーソル中心点水平座標}
151:     dos_reg.cx := centerY;    {カーソル中心点垂直座標}
152:     dos_reg.es := seg(shape);{カーソル形状データアドレス}
153:     dos_reg.dx := ofs(shape);
154:     Intr(MOUSE_INT,dos_reg)
155:   end;
156: {--------------------------------------------------}
157: {ファンクション 11}
158: procedure Movement (var moveX,moveY : integer);
159: {マウス移動距離の取得}
160:   var
161:     dos_reg : Registers;
162:
163:   begin
164:     dos_reg.ax := 11;
165:     Intr(MOUSE_INT,dos_reg);
166:     dos_reg.cx := moveX; {水平方向移動距離}
167:     dos_reg.dx := moveY  {垂直方向移動距離}
168:   end;
169: {--------------------------------------------------}
170: {ファンクション 12}
171: procedure Set_CaLL (caLL : word; RTSeg,RTOfs : word);
172: {ユーザ定義ルーチン呼び出しセット}
173:   var
174:     dos_reg : Registers;
175:
176:   begin
177:     dos_reg.ax := 12;
178:     dos_reg.cx := caLL;       {コール条件}
179:     dos_reg.es := RTSeg;{ユーザ定義ルーチンアドレス}
180:     dos_reg.dx := RTOfs;
181:     Intr(MOUSE_INT,dos_reg)
182:   end;
183: {--------------------------------------------------}
184: {ファンクション 15}
185: procedure Set_Mickey_Dot (ratioX,ratioY : integer);
```

```
186: {ミッキー・ドット比の設定}
187:   var
188:     dos_reg : Registers;
189:
190:   begin
191:     dos_reg.ax := 15;
192:     dos_reg.cx := ratioX;{水平方向比}
193:     dos_reg.dx := ratioY;{垂直方向比}
194:     Intr(MOUSE_INT,dos_reg);
195:   end;
196: {-------------------------------------------------------}
197: {ファンクション 16}
198: procedure Move_Set_X (minX,maxX : integer);
199: {水平方向移動範囲の設定}
200:   var
201:     dos_reg : Registers;
202:
203:   begin
204:     dos_reg.ax := 16;
205:     dos_reg.cx := minX;
206:     dos_reg.dx := maxX;
207:     Intr(MOUSE_INT,dos_reg);
208:   end;
209: {-------------------------------------------------------}
210: {ファンクション 17}
211: procedure Move_Set_Y (minY,maxY : integer);
212: {垂直方向移動範囲の設定}
213:   var
214:     dos_reg : Registers;
215:
216:   begin
217:     dos_reg.ax := 17;
218:     dos_reg.cx := minY;
219:     dos_reg.dx := maxY;
220:     Intr(MOUSE_INT,dos_reg);
221:   end;
222: {-------------------------------------------------------}
223: {ファンクション 18}
224: procedure Set_PLane (pLane_no : integer);
225: {表示画面プレーン設定}
226:   var
227:     dos_reg : Registers;
228:
229:   begin
230:     dos_reg.ax := 18;
231:     dos_reg.bx := pLane_no;{プレーン番号}
232:     Intr(MOUSE_INT,dos_reg)
233:   end;
234: {-------------------------------------------------------}
```

```
235: {ユーザ定義ルーチン}
236: {$S-,F+}
237: procedure TestRt;interrupt;
238: begin
239:   InLine ($FA);                 { CLI }
240:   Write(^g);
241:   {ルーチン終了処理}
242:   InLine (
243:     $89/$EC/    { MOV SP,BP }
244:     $FB/{ STI }
245:     $5D/{ POP BP }
246:     $07/{ POP ES }
247:     $1F/{ POP DS }
248:     $5F/{ POP DI }
249:     $5E/{ POP SI }
250:     $5A/{ POP DX }
251:     $59/{ POP CX }
252:     $5B/{ POP BX }
253:     $58/{ POP AX }
254:     $CB { RETF }
255:   );
256: end;
257: {$S+,F-}
258: {=========================================================}
259: {デモ用メインプログラム}
260: var
261:   dumc,dumx,dumy : integer;
262:   cdata : CursorForm;
263:
264:   begin
265:
266:     if InitialL = 0 then
267:       begin
268:         WriteLn('マウスが使用できません。');
269:         HaLt
270:       end;
271:
272:     for dumc := 0 to 31 do cdata[dumc]:=$FFFF;
273:     CLrScr;
274:     WriteLn(' ***** MOUSE DEMONSTRATION ***** ');
275:     Write('右ボタン押下で終了します。');
276:     TextCursor(NoDispCursor); {テキストカーソルを隠す}
277:     Set_Shape(7,15,cdata);    {カーソルを四角にする}
278:     Set_Position($013b,$00c0);{ 中点セット }
279:     Indication;               { マウスカーソル表示 }
280:     Move_Set_X($0000,$0275);  { X 軸セット }
281:     Move_Set_Y($0000,$0180);  { Y 軸セット }
282:     Set_CaLL($02,Seg(TestRt),Ofs(TestRt));
283:                               {左ボタン押下でユーザ定義ルーチン}
```

```
284:
285:     repeat  untiL Right_Press(dumc,dumx,dumy)=-1;{右ボタン押下}
286:      dumc := Initial;
287:      CLrScr;
288:      Erase;                                { マウスカーソル消去 }
289:      WriteLn(' ***** MOUSE DEMONSTRATION END ***** ');
290:      TextCursor(DispCursor);            {テキストカーソル復活}
291:    end.
```

リスト6-1を見てください。マウス機能をサポートしている部分は5～195行です。本来この部分はユニット・ファイルにしてあなたのライブラリにして、マウス利用のプログラムを作るときに使いやすくしておくとよいでしょう。もしそうしたければ、TURBOエディタを使います。エディタでこのリストを入力したら、Ctrl-QRと押してカーソルを先頭に設定します。次にCtrl-KBと押してマーク開始にしてから、カーソルを195行のend；の後ろに移動します。

そこでCtrl-KKと押すとマウス機能の部分がブロック指定でカラーが変わります。Ctrl-KWを押すとファイル名を聞いてきますから、MOUSE.PASとすれば、カレント・ドライブにMOUSE.PASファイルが作られます。こうしておいてCtrl-KYとするとブロック指定された部分が消滅します。3行目のusesにmouseをユニットとして加えます。

むろん、MOUSE.PASとして作ったファイルはユニットとしての形式を第4章を参照して整えなければなりません。ユニットにしておくとマウス利用のプログラムの開発がぐんと楽になりますので、ぜひユニット化をおすすめしておきます。

余談ですが、このCtrl-KK、KB、KWでプログラムの一部をフロッピーに格納する方法は便利です。たとえば、いまエディットしているTURBO Pascalのプログラム・ファイルがABCD.PASのときに、これを改造したプログラムを作ると同じ名前でしか格納できないので、改造前のプログラムをとっておけません。そこで、この方法を使うのです。ファイルメニューのSave as... でも同じことが可能ですが、それより手軽です。

エディタで改造したABCD.PASの先頭をCtrl-KB、最後をKKとしてからCtrl-KWとしてたとえばKAIZOU.PASとすれば、改造したABCD.PASをKAIZOU.PASとしてフロッピーに格納することができます。Ctrl-KDでエディタを抜けて、セーブするかどうかの質問にN(ノー)を返せばよいのです。このテクニックはいわばコロンブスの卵ですが、とても役に立つので、憶えておくとよいでしょう。

さて、リスト6-1を眺めてみましょう。例によって割り込みを使用するので2行目でCrtだけでなくDosユニットの使用を宣言しておきます。これにより内部レジスタにアクセスするためのレコード変数型Registersを使用できます。

5行目はマウス割り込みベクタを$33とし定数定義しています。この情報はmouse.sys付属の資料によりました。つまり、マウスになんらかの操作が加えられるときに、$33

番の割り込みによってそれを知るようになっています。8 行目に定義されている型、CursorForm は後でマウスカーソルの形状を作るときのビット情報データを入れる変数を作るときに使います。

マウスの処理のための関数、手続きは 11 行目から始まります。マウスドライバは 0 ～18 までの 16 個のファンクション (機能) を実行できます。0 ～ 18 なら合計 19 個のはずですが、ファンクションの 10、13、14 は欠番です。ファンクション番号を A レジスタにロードして割り込みをかけると、各種機能が実行できます。

11 行目はマウス環境のチェックの関数 Initial です。17 行目の A レジスタを 0 に設定しているのは$33 の割り込みを 18 行目で行う前に、その割り込みがマウス環境のチェックであることをデバイス・ドライバ mouse.sys に知らせる役目があります。この方式は以下の各手続き、関数で共通しています。チェックの結果は 18 行目の割り込みの結果としてレジスタに返されます。A レジスタが 0 ならマウスが使用できない環境、－1 ならできる環境です。その値をこの関数 Initial の値として返します。

このファンクション 1 の初期化によって次のようにデフォールトが設定されます。

カーソル表示	表示しない
カーソル位置	スクリーンの中心位置
カーソル形状	左上向き矢印
表示画面	プレーン 2
カーソル中心点	(0,0)
移動範囲	スクリーン全体
ミッキー/ドット比	水平、垂直方向とも 8
マウス割り込み周期	8ms

23 行目の手続き Indication はマウスカーソルを画面に表示させる手続きです。ファンクション 0 か次のマウスカーソルを消去するファンクション 2 を実行するまで有効です。34 行目の手続き Erase は、表示されているマウスカーソルを画面に表示させなくする手続きです。これはファンクション 1、つまり Indication を実行するまで有効です。また、Erase を複数回使用した場合は、その回数と同一回数 Indication でファンクション 1 を実行しないとマウスカーソルは表示されません。よくこの手続きを何度も呼び出してマウスが「行方不明」になることがありますから、何度呼んだのかをきちんと追跡しておかねばなりません。

45 行目はファンクション 3 の関数 Position です。マウスのボタンの状況とマウスカーソルの現在の表示位置を取得する関数です。54、55 行目でボタンの状態が A、B、C、D レジスタに出力されます。C、D レジスタのマウスカーソルの表示位置は画面の左隅を座標原点とし、右に向かって X 座標 (0 ～ 639)、下に向かって Y 座標 (0 ～ 399) になります。A、B レジスタに返されるボタンの状態は次のようになっています。

AX＝左ボタンの状態

0：左ボタンが離されている。

－1：左ボタンが押されている。

BX＝右ボタンの状態

0：右ボタンが離されている。

－1：右ボタンが押されている。

この状態を関数値として外へ取り出すために、54 行目で 10 進数の 10 の位に左ボタンの状態、1 の位に右ボタンの状態を入れています。これはあまりスマートではないので、論理変数を使ってもっとかっこよく作れるはずですが、それは読者におまかせしておきます。

61 行目はマウスカーソルの位置を設定する手続きで、ファンクション 4 になります。各座標の設定値の範囲は、水平座標が 0 ~ 639、垂直座標が 0 ~ 399 です。これをこえる範囲を指定したときには、移動範囲内の端にマウスカーソルを移動します。

74 行目の関数、Left_Press と 91 行目の関数、Left_Release はマウスのボタンの押下と解放に関する情報を取得する関数です。これがファンクション 5 と 6 になります。むろん、この 2 つの関数をひとつにまとめて左ボタンの押下・解放情報をえるための関数を自作してもよいでしょう。84 行目は左ボタン押下時に－10 、解放時には 0 に関数値をするためのものです。85 行目の count にはこの関数が最後に呼ばれてから、今回呼ばれるまでに左ボタンが押された回数が入ります。LastX と LastY には最後の左ボタン押下時の座標値です。91 行目の関数 Left_Release は Left_Press の押下と解放を逆にしたものです。

108 行目の関数 Right_Press と 125 行目の関数 Right_Release は上の左ボタン用の関数を右ボタン用にしたものです。

142 行目の手続き Set_Shape はマウスカーソルの形状を設定します。ファンクション 9 になります。カーソルの形は、横 1 ワード (16 ビット) 、縦 32 ワードのデータのドットパターンで表します。32 ワードのうち始めの 16 ワードのパターンでグラフィック画面と AND をとり、次の 16 ワードで XOR をとります。したがって、後の 16 ワードより 1 回り大きい反転パターンを先の 16 ワードに設定することによって、カーソルを黒い縁で枠付けすることができます。また、カーソルの中心とはカーソルの左上隅を (0,0) とした座標系で与えます。

カーソル形のデータ例

$$\left.\begin{array}{ll} DW & 0011111111111111B \\ DW & 0010111111111111B \\ & \vdots \\ DW & 0000000000000000B \end{array}\right] \text{32 行}$$

158 行目の Movement は、マウスの移動距離を取得する手続きで、ファンクション 11 です。この手続きが最後に使用されてからのちマウスが動かされた距離が得られます。距離の単位はミッキーなのでファンクション 15 によって設定されたミッキー/ドッ

ト比によって画面でカーソルが動いた距離と現実にマウスが机上を移動した距離とは異なった値が得られます。ミッキーとはマウスの物理的な移動距離を表す単位です。1ミッキーは約1/100インチ(約0.25mm)です。この単位は「ミッキーマウス」からの連想でしょうか。カーソルの動いた距離ではなくマウスが実際に動いた距離ですから、カーソルが移動範囲の隅にあり動けない場合でもマウスが動くかぎり距離はカウントされます。

171行目のSet_Callはマウスの状態によりユーザーが記述したサブルーチンを呼び出し(コールし)て実行させるための条件を設定するものです。コールする条件とサブルーチンの入口アドレスを設定する手続きです。これがファンクション12です。178行目のコール条件には下記の5つがあります。

ビット0＝カーソル位置の変化
1＝右ボタンが押される
2＝右ボタンが離される
3＝左ボタンが押される
4＝左ボタンが離される
5～15＝未使用

（ビット0～4について）最下位ビットを0にして、各ビットが1のときその条件発生でコール要求

コール条件が発生するとドライバはセグメント外サブルーチンコール命令を使ってユーザーサブルーチンをコールしますから、そのサブルーチンから再び戻るときにはセグメント外リターン命令を使用します。また、このSet_Callはレジスタに下記の情報を用意してユーザーのサブルーチンをコールします。

AX＝コールの原因となった現象、CXに設定するパラメータと同じ。

ビットが1になっている現象がコール原因となった

BX＝ボタンの状態

ビット0＝0：左ボタンが離されている。
1：左ボタンが押されている。
1＝0：右ボタンが離されている。
1：右ボタンが押されている。

CX＝カーソルの水平座標

DX＝カーソルの垂直座標

185行目のSet_Mickey_Dotは、マウスの物理的な移動距離ミッキーとそれに対するカーソルの移動ドット数と比を設定する手続きです。水平、垂直方向それぞれに対して、カーソルが8ドット移動するのに要するマウスの移動量をミッキーの単位で設定します。ただし、0に設定すると何もしないで戻ってきます。これがファンクション15です。

198行目の手続きMove_Set_XとMove_Set_Yはマウスカーソルの移動範囲をそれぞれ水平方向(X方向)と垂直方向(Y方向)に設定するためのものです。むろん、水平方

向は0～639、垂直方向は0～399の範囲内となります。

最後の224行目の手続きSet_Planeはマウスカーソルをグラフィック画面の何番プレーンに表示するかを決めます。Bレジスタにカーソルの表示プレーンとして0～3を指定します。各画面の初期値はプレーン0から順番に青、赤、緑、灰色となります。

以上がマウス・ドライバmouse.sysを使用したTURBO Pascalのマウス機能ルーチンです。これを利用すればほとんどのマウス動作をサポートできるはずです。もし、もっと特殊なことをやらせたければ先ほどのファンクション12のSet_Callを使えばよいでしょう。

その例として235行目からユーザー定義ルーチンを作ってあります。ユーザー定義ルーチンは割り込みプログラムになりますので、少々特殊な注意が必要です。詳しくは日本におけるTURBO Pascalの販売を行なっているMSA(マイクロソフトウェア・アソシエイツ)社が頒布ディスクに入れてくれているマウス用のサンプルプログラムを参考にしてください。TURBO Pascalディスクには他にもいろいろと参考になるサンプルプログラムが豊富に入っているので、とても勉強になります。

さて、そのサンプルプログラムを参考にして作ったのがユーザー定義ルーチン、TestRtです。まず、スタックのチェックを止めさせ、発生するコードがセグメント外コールを使用するために236行目のコンパイラ指令を入れておきます。239行目で割り込み禁止にします。このルーチンの内容自体は240行目のブザー鳴動のみです。つまり、このルーチンは呼ばれるたびにブザーを鳴らします。

242～255行目までのInline文はマウスドライバと組み合わせてユーザー定義ルーチンを作るときには必ずこの終了法を使用します。この情報は先ほどのMSAのデモプログラムから手に入れました。この内容はTURBO Pascalの手続きがどのように引数を渡したり、レジスタを保存するのかがわからないと何をやっているのかよくわからないはずです。ここではあまり気にしないでこのとおりをまねしておけばよいでしょう。この形式でユーザー定義ルーチンを終了しないと、ほとんどの場合暴走するか、運よく正常に終了したかに見えてもMS-DOS自体がまともに動かなくなってしまいます。

こうして作り上げたマウス機能用の関数と手続き、そしてユーザー定義ルーチンの使用法をデモするのが259行目からのメインプログラムです。261行目の整数型変数はマウスボタンの押下情報を手に入れるための関数呼び出し時にダミー(どうでもよい)の引数として必要なものです。

262行目のcdataは8行目の型宣言で32個のword型の配列であるCursorForm型としています。これは後でマウスカーソルの形状を指定するためのものです。

まず、266行目で関数Initialを呼び出して、マウスが使用できる環境かどうかをチェックします。この関数が0であるときにはマウスが使えないので、プログラムを終了します。

272行目でマウスの形状としてフルの四角形を設定します。むろん、ここでビットを

構成してネズミの形にしても決して悪くはないのですが、そこまでの遊びは実例を示すことには直接関係ないのでやめておきました。プログラムの以下の部分では右ボタン押下で停止、左ボタン押下でブザーがユーザー定義ルーチンにより鳴動するようにしています。

276行目でDosユニットの手続きであるTextCursorによりテキストカーソルを画面から隠しておきます。こうしておかないと、マウスのデモ中に画面で四角いテキストカーソルがチカチカ点滅するので目障りになります。

277行目でカーソルの形状をcdataにもとづき四角に設定し、278行目でそれを画面の中心に位置させておきます。もっとも、これは最初のInitialによりデフォルトでそうなっているので、必ずしも必要ありません。

279行目のIndicationにより画面にマウスカーソルが現れます。280、281行目で画面全体をマウスカーソルが移動できるように範囲設定を行ないます。

282行目のSet_Callは第1パラメータが$2、つまり第1ビットが立っていますから、左ボタンが押下されると第2、第3パラメータでセグメントとオフセットが指定されているユーザー定義ルーチンTestRtを呼び出します。つまり、左ボタン押下でブザー鳴動します。

285行目は右ボタン押下までプログラムを繰り返すためのものです。右ボタンが押下されるとRight_Presss関数値が−1になるのを利用しています。

287行目以下は終了のための処理です。ClrScrにより画面のテキストを消去して、Eraseによりマウスカーソルを画面から消し去ります。最後に290行目でDosユニットのTextCursor手続きにより隠してあったテキストカーソルを復活します。

こうして作ったプログラムを走らせるときには、その前に、上に説明したようにディスクの中にmouse.sysがあるか確かめ、mouse.sysをconfig.sysファイルに加えてます。つまり、device = mouse.sysの1行を加えるのです。

以上でプログラムの解説は終りです。いままでどうやってマウスを制御していたのか不思議に思っていたことが解決したのではないでしょうか。ここで紹介したマウスの機能のほとんどは前述のディスクの中に収められていますが、TURBO Pascalのソースコードとして記述したリスト6-1のサンプルプログラムのほうがどうなっているのかよくわかるでしょう。むずかしいのはユーザー定義ルーチンの作り方ですが、マウスの条件によりいろいろなことをやるルーチンを作ってみて実験してください。

7 オブジェクト指向でグラフィックス

パソコンでプログラムを作る楽しさはなんといってもパソコンの強力なグラフィック機能の活用にあります。しかし、ここに大きな問題があります。グラフィック機能はパソコンの個性そのものですから、パソコンの機種ごとに異なる仕様になっています。同じ図形を描くのでも微妙なところが違うので、標準の Pascal でもグラフィックスについては何も命令がありません。ですから、標準の Pascal ではパソコンの機能を十分に引き出すことはできません。

ところが、TURBO Pascal はいかにパソコンのプログラミングをサポートするかを第一に設計されたコンパイラですから、グラフィックスについても簡単にプログラミングできるように十分配慮されています。TURBO Pascal の ver4.0 以上では BGI(Borland Graphics Interface：ボーランド・グラフィックス・インタフェース)が付属しています。このグラフィックスドライバを利用することにより、多彩なグラフィックスをプログラムに組み込むことが可能です。

BGI はプログラム上には直接現れません。プログラム的には TURBO Pascal に付属してくる Graph.tpu というユニットを uses 宣言して、58 種の手続きと 22 種の関数によりグラフィックスを実現します。しかし、この Graph.tpu を使用したときには、必ず BGI を組み込んでコンパイルしないとプログラムが動作しません。

具体的な組み込み手順は本書の姉妹編である「TURBO Pascal トレーニングマニュアル」に基本的なことが書いてありますが、この章ではグラフィックス活用の観点からもう少し突っ込んだ解説をします。さらに、グラフィックスをプログラミングする上でのテクニックもいくつかの例題を通して解説します。

なつかしのゲームに挑戦

7–1■アニメーション

グラフィックスとは要するにコンピュータで絵を描くことです。プログラムを作り始めたころはスクリーンに丸や四角をいろいろなカラーで描いているだけでもうれしいものですが、それではすぐに飽きてしまいます。飽きてしまう最大の理由は描いた図形が動かないことです。

むろん、図形が動かなくてもすばらしいグラフィックスが作れます。そちらのほうはこれも本書の姉妹編である「ターボグラフィックス」に詳しく解説されています。ここではまずアニメーションを取り上げてトレーニングします。パソコンのアニメーションはわかってしまえば簡単な原理を使用しています。もしもあなたがおもしろいゲームを作ってみようというの欠かすことのできないテクニックです。

画面上で図形を動かすには次の順序で動いているように見せるのが常套手段です。

(1) 図形を描く

(2) 図形を隠す

(3) 数ピクセル (画素) ずれたところに図形を描く

(4) (1) ~ (3) を繰り返す

(1) で画面に現れた図形が (2) で消されても、その図形は目の中に残像としてしばらく残っています。(3) で残像のすぐ近くに同じ図形を描かれると、目には図形が移動したように見えます。これを繰り返せばアニメーションができるわけですが、動きのなめらかなアニメーションを作るにはさらに少々テクニックが必要です。

理想的をいえば、(3) で 1 ピクセルだけずれたところに図形を描けば、一番動きがなめらかになるはずです。画面の図形の最小単位が 1 ピクセルですから、それ以下に細かく図形を移動することは物理的、つまりハード的に無理な相談です。それならそうすればよいではないかということになりますが、そうはいかない事情があります。パソコンの処理速度との関係で、1 ピクセルずつの移動ではアニメーションが極度に低速になってしまうからです。その対策として、不自然に見えない範囲内でぎりぎりのピクセル数で移動させてやれば、アニメーションの速度は向上します。

こうしたとしても、まだ問題があります。動かそうとしている図形がかなり複雑だと、画面上でその絵を描いている過程が見えてしまいます。むろん、CPU の処理速度がかなり速ければ、複雑な図形でも一瞬に描くことができます。しかし、現在のパソコンでは円でさえも、ちょっと大きくなると描くのが見えてしまいます。

以上のことを頭に入れて、リスト 7-1 を見てください。この例ではなつかしのパックマンもどきを画面上で動かします。

■リスト 7-1

```
 1: program LIST7_1;
 2:
 3: uses Crt,Graph;
 4:
 5: Var
 6:   GraphDriver,GraphMode : integer;
 7:   active, visual, temp : word;
 8:   XCenter,YCenter, Radius, StAngle, EndAngle : integer;
 9: {---------------------------------------------------------}
```

```
10: {グラフィックドライバの初期化}
11: procedure Initialize;
12: begin
13:   GraphDriver := detect;
14:   InitGraph(GraphDriver,GraphMode,'');
15:   if not(GraphDriver in [PC98,PC98HI]) then
16:     begin{グラフィックドライバがない}
17:       RestoreCrtMode;
18:       WriteLn('グラフィックドライバがありません。');
19:       Halt
20:     end;
21:   SetColor(Red);
22:   SetFillStyle(SolidFill,Blue);
23:   XCenter := 10;
24:   YCenter := GetMaxY div 2;
25:   StAngle := 0;
26:   Radius := 10;
27:   active := 0;
28:   visual := 1
29: end;
30: {-------------------------------------------------------------}
31: {パックマンを描く}
32: procedure Draw_Pacman;
33:
34: begin
35:   ClearViewPort;
36:   if StAngle = 0 then
37:    begin
38:       StAngle := 30;
39:       EndAngle := 330;
40:    end
41:   else
42:    begin
43:       StAngle := 0;
44:       EndAngle := 360;
45:    end;
46:   PieSlice(XCenter,YCenter,StAngle,EndAngle,Radius)
47: end;
48: {-------------------------------------------------------------}
49: {メインプログラム}
50: {-------------------------------------------------------------}
51:
52: begin
53:   Initialize;
54:   while not KeyPressed do
55:     begin
56:       SetActivePage(active);
57:       SetVisualPage(visual);
58:       Draw_Pacman;
```

```
59:      temp := active;{画面切り替え}
60:      active := visual;
61:      visual := temp;
62:      XCenter := XCenter + 10;
63:      if XCenter>649 then
64:         begin{新しいパックマン生成}
65:           XCenter :=10;
66:           SetColor(Random(GetMaxColor)+1);
67:           SetFillStyle(SolidFill,Random(GetMaxColor)+1);
68:         end
69:    end;
70:  CloseGraph{グラフィックス使用終了}
71: end.
```

3行目でCrtとGraphユニットの使用を宣言しておきます。5行目の変数宣言では特に6行目のグラフィックスドライバを初期化するときの定石のための変数に注意してください。7、8行目の変数はプログラムに現れたところで説明します。

まず、11行目から始まる手続きInitializeです。この手続きで大切なのはグラフィックス使用のための準備手順です。13行目では、使用するグラフィックスドライバをパソコンのグラフィックスハードに合わせたものとするため、Graphユニットの定数、detectにしています。こうすると、PC-9801の機種によりノーマル表示にはPC98、ハイレゾ表示にはPC98HIのいずれかのグラフィックスドライバを使用できるようになります。

こうしておいて14行目でInitGraphを呼び出すと、detectにより使用しているパソコンに応じてグラフィックスドライバをセットし、グラフィックスのモードを640 × 400ドットの8色モードまたは16色モードにセットします。3つめのパラメータの" はグラフィックスドライバ、BGIを拡張子に持つファイルカレントディレクトリ上からさがすという指定です。これがたとえばSAMPLEというディレクトリ上になるのなら、ここは'¥SAMPLE'とします。

InitGraphから返されたドライバがPC-9801のグラフィックス用ハードに適合するかどうか15行目で調べます。グラフィックドライバがなければここでエラーメッセージを出力してプログラムを停止します。17行目で手続きRestoreCrtModeを呼び出していますが、PC-9801はグラフィックスとテキストの同時表示が可能ですから、これは必要ありません。このプログラムをIBM-PC互換機で動かすときには必要です。21行目以降はグラフィックドライバの組み込みに成功してから実行されます。

21行目でこれから描く図形の線を赤に指定し、図形の内部をフィルするときにはベタ塗りすることを指定します。23、24行目でこれから描くキャラクタの中心点を画面左端の真ん中にします。46行目でキャラクタを描く具体的な方法として扇形グラフ描画の手続きPieSliceを使用しています。そのための出発角を25行目で0度にしています。26行目はキャラクタの大きさで、半径10ピクセルにしています。

27 、28 行目で変数 active と visual にそれぞれ 0 と 1 を入れています。ここでは PC-9801 の GVRAM プレーン 0 番と 1 番を利用します。active で指定されたプレーンに絵を描き、visual で指定されたプレーンを表示するようにしています。実はこのプログラムではプレーン 0 と 1 を交互に表示させることにより、キャラクタを描いている過程が外から見えないようにしているのです。

32 行目の手続き、Draw_Pacman では Graph.tpu で提供されている手続き PieSlice によりキャラクタが口を開けているところと閉じているところを描きます。まず、35 行目で以前に描いたキャラクタを消してから、開始角度の違いにより描き分けます。

36 行目では以前に描かれたキャラクタは口を閉じていますから、今度は扇形を 30 〜 330 度に描き、口を開いている形にします。そうでなければ口を開いていたのですから、41 行目以下で 0 〜 360 度で口を閉じた形にします。こうしておいて 46 行目でキャラクタを描きます。このとき、Initialize 手続きで線を赤、フィルを青のソリッドにしてありますから、輪郭が赤で中が青いキャラクタが描かれます。

52 行目からがメインプログラムです。54 〜 69 行目をなにかキーが押されるまで続けます。56 行目でグラフィックス出力の対象、つまり描画の対象になる画面を決めます。最初は 27 行目によりプレーン 0 番が描画対象です。57 行目でプレーン 1 番を画面表示の対象にしています。58 行目で手続き Draw_Pacman を呼び出すと、描画対象画面であるプレーン 0 番に口を閉じたキャラクタが描かれます。しかし、表示対象画面はプレーン 1 番なので、それは表示されません。

こうしておいてから、59 〜 61 行目で描画対象と表示対象のプレーン番号を交換しておきます。62 行目でキャラクタ、つまり扇形の中心の座標を右へ 10 ピクセル進め、63 行目で X 座標が画面の左端に行ったところで新しいキャラクタを作ります。新しいキャラクタは画面左の 10 ピクセル目が中心で、輪郭や中の色は乱数で決めています。

キーを押下しない限り再び 54 〜 69 行目が実行されます。56 、7 行目で描画対象画面と表示対象画面が今度はそれぞれプレーン 1 と 0 になっています。そこで手続き Draw_Pacman を呼び出すと、さきほどプレーン 0 に描いておいた口を閉じたキャラクタが表示されます。そしてプレーン 1 には口を 60 度開いたキャラクタが描かれます。つまり、これを右へ 10 ピクセルずつ進めながら交互に表示することにより、描画過程を画面では見せないでなめらかに移動しているように見せかけているわけです。70 行目でグラフィックスの使用を終了してプログラムを終わります。

もしもあなたのパソコンではキャラクタの移動が速すぎるときには 62 行目の移動速度を 10 ピクセル以下にしてみてください。1 ピクセルにしても速すぎるようだったら 62 行目と 63 行目の間に適当な時間遅れを発生する Delay 関数を入れます。

いずれにしても、ここの例で示したように Graph ユニットを使用するときには、13 行目からの一連の手続きを定石として使うことを覚えておいてください。もういちどまとめると、以下のようになります。

```
...
...
GraphDriver := detect;
InitGraph(GraphDriver,GraphMode,'');
if not(GraphDriver in [PC98,PC98HI]) then
  begin
    グラフィックスドライバがないときのエラー処理;
    Halt;
  end;
...
...
```

上の if 文は InitGraph の結果を返す関数 GraphResult と、Graph ユニットの定義済み定数 GrOK を利用して、次のように書いても同じです。

```
if GraphResult <> GrOK then
```

ここで紹介した GVRAM プレーンの切り替えは簡単な割に効果的なアニメーション効果をえられるテクニックです。もっとも PC-9801 の初期のモデルや U2 のような古い機種だとグラフィックスページ (GVRAM) は 1 画面分しかありませんから、このテクニックは使えません。

ビリヤードの玉を動かす

7–2■オブジェクトアニメーション

7–2–1■オブジェクトを理解する

この章がグラフィックスに関するところなのに、なぜオブジェクトの話が出てきたのか不思議に思われるかもしれません。オブジェクトについては、なんだかよくわからないが、とにかくコードとデータを一体化するプログラミング手法なのではないのかというのが一般的な理解ではないでしょうか。

オブジェクトが単なる概念的なものでしかないのであれば、何よりも実用性を貴んでいる TURBO Pascal に取り込まれたはずがありません。ver5.5 でオブジェクトが初めて導入されたときにはなんのためかはっきりしなかったのですが、ver6.0 ではオブジェクトを積極的に使用したプログラム開発環境が整えられています。たとえば、ver6.0 に付加された TURBO Vision(ターボビジョン) はすべてオブジェクトを利用しています。

そこで、ここではオブジェクトの完全理解とアニメーションをドッキングしたトレーニングをやってみます。TURBO Pascal のマニュアルセットにオブジェクト指向プログラミング (Object Oriented Programming) についてかなり詳しい解説があります。しかし、この解説は厳密さを大切にしているためか、少々「お固く」て読んでもなかなか

ピンとはきません。ぼくも何度も何度も読み返し、いろいろと試行錯誤した挙げ句にやっと理解できました。もっとも、それはぼくの頭が悪いだけで読者のあなたにはやさしかったかもしれません。

TURBO Pascal のマニュアルではオブジェクトについて継承性 (Inheritance：インヘリタンス)、カプセル化 (Encapsulation：エンキャプシュレーション)、そして多態性 (Plymorphism：ポリモーフィズム) という 3 つの基本的性質から理解するという正攻法を取っています。

それはそれで厳密性を大切にするマニュアルとしては正しいのですが、オブジェクト指向プログラミングを理解しがたい難しいものだという印象を与えていることも事実です。そこで、ここでは例題プログラムを通して要するに TURBO Pascal のオブジェクト指向プログラミングをどう利用してプログラムをきれいに素早く書くかという実戦面を中心に解説することにします。

さて、例としてビリヤード・テーブル上にボールがいくつか投げ出されたときのアニメーションをプログラミングしてみます。このプログラムを作ろうとしたときに、まず困るのはビリヤードテーブルやボールをどう表現するかです。そこでオブジェクト指向プログラミングが威力を発揮します。

オブジェクト指向プログラミングでは頭の中に浮かんだ「モノ」をオブジェクトとしてとらえます。このような発想法は今までのプログラミングのそれとは違うためにかなりの違和感があり、オブジェクト指向プログラミングはわからないと投げ出されてしまう最大の原因になっています。

この例を頭の中に想像すると、ビリヤードテーブルの中をお互いにぶつかり合ったりしながら動き回るボールが「モノ」＝オブジェクトとして浮かんできます。むろん、ボールが動き回る環境を提供している「モノ」としてのビリヤードテーブルもありますが、ビリヤードテーブルはなんらかの主体性を持っているものではなく、ボールに対して動き回るための環境を提供しているだけです。テーブルがボールに与える摩擦や、ボールの運動範囲を制限するテーブルの大きさや、運動方向に影響するテーブルの傾きなどを考えてください。そのどれもボールというオブジェクトに影響を与えると考えたほうが自然ではありませんか。

ここまでで大切なのは、何がオブジェクトなのか、そしてそのオブジェクトが存在する環境とはどのようなものかをはっきり分けて考えるということです。ここではオブジェクトを画面に表示されるボールとして、環境は画面全体とします。つまり、画面全体を環境としてオブジェクトであるボールが複数個運動することになります。

ここまで考えてからいよいよオブジェクト指向プログラミングでプログラムを作ります。オブジェクトとしてのボールにはどのような属性、つまり性質があるのでしょうか。それを環境である画面との関係で考えると次のような属性があります。

リスト 7-2 の 17 ～ 22 行目までを見てください。

(X,Y)	:	画面上での位置
Radius	:	半径
Id	:	番号
(XDirection,YDirection)	:	移動方向
Visable	:	画面表示するかどうか
Image	:	形状データ
Image_Size	:	形状データの大きさ形状

ここまではあるボールが固有に持っている性質ですが、属性としてボールがどのように運動するかという、いわば動的な属性もあります。それが24～31行目までのメソッドです。つまり、TURBO Pascalのオブジェクト指向プログラミングでメソッドと呼ばれているのがオブジェクトの動的な属性なのです。この例のオブジェクト、ボールには次のようなメソッドがあります。

Init	:	環境内に生れる
Done	:	環境から消滅する
PutImage	:	画面に現れる
Collisions	:	他のボールと衝突する
Bounce	:	環境と衝突する

以上をまとめると、オブジェクトの静的および動的な属性として何があるのかを並べてしまうことが先決ということになります。あとは異なる属性を持ったボールを複数個作って環境の中に「放流」してやれば、ボール同士が衝突したり環境の端、つまり画面の端から跳ね返ったりするはずです。こうして、ボールの静的属性であるデータと動的属性であるコードが一体化していることこそカプセル化にほかなりません。

残る問題はボールを複数個どうやって作るかです。作るボールの個数が決まっているのならプログラム中にボールオブジェクトのインスタンスとして個数だけボールを作ればよいので簡単です。それではおもしろくありません。1個以上の好きな個数のボールを作れるようにしてみましょう。

データベースなどで、ある複数のレコードが並んでいる構造に自由に追加、削除するときにはどうすればよいのか考えてみてください。動的リンク構造を使用すればよいことに気がつきましたか。Pascalプログラマが得意(苦手?)とするポインタを使用して、データの連鎖としてヒープ上にレコードを作るのは、レコード個数が不定のときに使うテクニックです。実は、TURBO Pascalでオブジェクトがレコード型に類似した構造を取っているのはこのためです。

■リスト 7-2

```
 1: program LIST7_2;
 2:
 3: uses
 4:    Crt,DOS,Graph;
 5:
 6: type
 7:    ByteArray = array[0..0] of byte;{イメージデータ用}
```

```
 8: ByteArrayPtr = ^ByteArray;{イメージデータポインタ}
 9:
10:   BPointer      = ^BaLL;    {メソッドのポインタ}
11:   BaLLList = ^BaLLSequence;{ボールリスト}
12:   BaLLSequence   = record   {ボールリストの構造}
13:                NextBaLL:BaLLList;
14:                ThisBaLL:BPointer;
15:             end;
16:
17:   BaLL=object{ボールオブジェクト}
18:        X,Y,Radius,Id                : integer;
19:        XSpeed,YSpeed,CoLor,Pattern  : integer;
20:        XDirection,YDirection,VisabLe : booLean;
21:        Image                        : ByteArrayPtr;
22:        Image_Size                   : word;
23:
24:        Constructor Init(InitX,InitY,InitRadius,
25:                         InitSpeed,InitAngLe,
26:                         InitCoLor,InitPattern,
27:                         InitId : integer);
28:        Destructor Done;
29:        procedure PutImage;
30:        procedure CoLLisions(CurrentBaLL : BaLLList);
31:        procedure Bounce(CurrentBaLL : BaLLList);
32:     end;
33:
34: var
35:   Head,TaiL : BaLLList;{リスト構造の頭と尾}
36:
37: {---------------------------------------------------------}
38: {ボールメソッドの定義}
39: {---------------------------------------------------------}
40: {ボールの生成}
41: Constructor BaLL.Init(InitX,InitY,InitRadius,
42:                       InitSpeed,InitAngLe,
43:                       InitCoLor,InitPattern,
44:                       InitId:integer);
45: var
46:   Rspeed : reaL;{作業用実数}
47:
48: begin
49:   Id := InitId;{ボール番号}
50:   X  := InitX;{初期X座標}
51:   Y  := InitY;{初期Y座標}
52:   Radius := InitRadius;{ボール半径}
53: {境界にかからないよう修正する}
54:   if (X < Radius) then X := Radius;
55:   if (Y < Radius) then Y := Radius;
56:   if (X > GetMaxX-Radius)
```

```
57:         then X := GetMaxX - Radius;
58:   if (Y > GetMaxY-Radius)
59:         then Y := GetMaxY - Radius;
60:   Rspeed := InitSpeed;
61:   XSpeed := Round(cos(InitAngLe)*Rspeed);{速度X成分}
62:   YSpeed := Round(sin(InitAngLe)*Rspeed);{速度Y成分}
63:   CoLor := InitCoLor;{ボールの色}
64:   Pattern := InitPattern;{ボールのフィルパターン}
65:   XDirection := (XSpeed>0);{X方向}
66:   YDirection := (YSpeed>0);{Y方向}
67:   XSpeed := Abs(XSpeed);
68:   YSpeed := Abs(YSpeed);
69: {ヒープ領域にボールビットイメージ領域確保}
70:   Image_Size := Graph.ImageSize(X-Radius,Y-Radius,
71:                                 X+Radius,Y+Radius);
72:   GetMem(Image,Image_Size);
73:   Graph.SetCoLor(CoLor);{ボールの色設定}
74:   Graph.SetFiLLStyLe(Pattern,Graph.GetCoLor);{フィルパターン設定}
75:   Graph.FiLLELLipse(X,Y,Radius,Radius);{ボールを描く}
76: {ビットイメージの取り込み}
77:   Graph.GetImage(X-Radius,Y-Radius,
78:                  X+Radius,Y+Radius,Image^);
79: {ボールを隠す}
80:   Graph.PutImage(X-Radius,Y-Radius,Image^,XORPut);
81:   VisabLe := FALSE;{画面表示オフ}
82: end;
83: {---------------------------------------------------------}
84: {ボールオブジェクトの消滅}
85: Destructor BaLL.Done;
86: begin
87:   FreeMem(Image,Image_Size);{ヒープ解放}
88: end;
89: {---------------------------------------------------------}
90: {ボールの画面表示}
91: procedure BaLL.PutImage;
92:
93: begin
94:   Graph.PutImage(X-Radius,Y-Radius,Image^,XORPut);
95:   DeLay(0);{表示時間：機種により調整}
96: end;
97: {---------------------------------------------------------}
98: {ボール衝突の処理}
99: procedure BaLL.CoLLisions(CurrentBaLL : BaLLList);
100: var
101:   XPos, YPos   : integer;
102:   XMine,YMine : integer;
103:   XSpd,YSpd    : integer;
104:   ORad         : integer;
105:   OXDir,OYDir : booLean;
```

```
106:
107: begin
108:   XMine := X;{ボールの位置}
109:   YMine := Y;
110:   CurrentBaLL := Head;{第一ボールから}
111:   whiLe(CurrentBaLL^.NextBaLL <> niL) do
112:     begin
113:       if (CurrentBaLL^.ThisBaLL^.Id <> Id) then
114:         begin{他のボールの方向、速度、半径、位置}
115:          OXDir := CurrentBaLL^.ThisBaLL^.XDirection;
116:          OYDir := CurrentBaLL^.ThisBaLL^.YDirection;
117:          XSpd := CurrentBaLL^.ThisBaLL^.XSpeed;
118:          YSpd := CurrentBaLL^.ThisBaLL^.YSpeed;
119:          ORad := CurrentBaLL^.ThisBaLL^.Radius;
120:          XPos  := CurrentBaLL^.ThisBaLL^.X;
121:          YPos  := CurrentBaLL^.ThisBaLL^.Y;
122:
123:       {衝突の判定}
124:          if (Abs(Xpos-XMine)<=(Radius+ORad)) and (Abs(Ypos-YMine)<=(Radius
+ORad))
125:           then
126:             begin{ボール衝突と方向チェック}
127:               Sound(100);
128:               DeLay(10);NoSound;;
129:               if ((YPos-YMine) <= 0) then
130:               {相手が上にある}
131:                 begin
132:                   if ((OYDir) and not(YDirection)) then
133:                      begin{上下反対方向の衝突}
134:                        YDirection := not(YDirection);
135:                        CurrentBaLL^.ThisBaLL^.YDirection := not(YDirection);
136:                      end
137:                    eLse
138:                      begin{上下同方向の衝突}
139:                        CurrentBaLL^.ThisBaLL^.YSpeed := YSpeed;
140:                        YSpeed := YSpd;
141:                      end;
142:                 end
143:             {相手が下にある}
144:               eLse
145:                 begin
146:                   if (not(OYDir) and (YDirection)) then
147:                      begin{上下反対方向の衝突}
148:                        YDirection := not(YDirection);
149:                        CurrentBaLL^.ThisBaLL^.YDirection := not(YDirection);
150:                      end
151:                    eLse{上下同方向の衝突}
152:                      begin
153:                        CurrentBaLL^.ThisBaLL^.YSpeed := YSpeed;
```

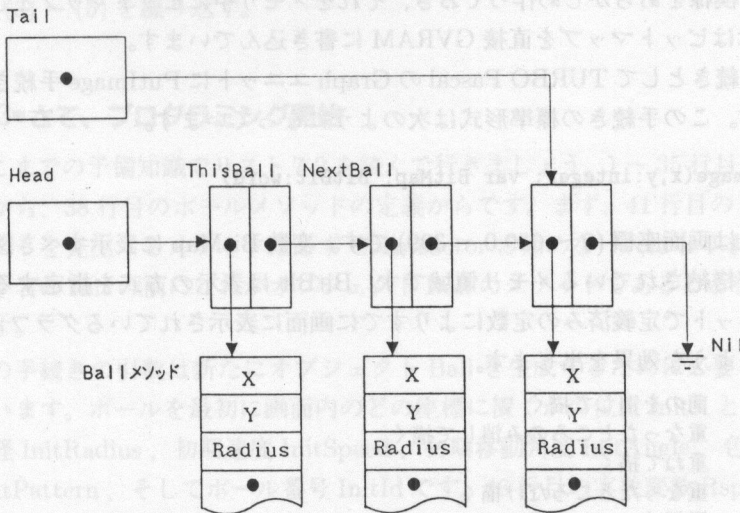
```
154:                    YSpeed := YSpd;
155:                  end;
156:              end;
157:
158:          if ((XPos-XMine) <= 0) then
159:              begin{相手が左にある}
160:                if ((OXDir) and not(XDirection)) then
161:                  begin{左右反対方向の衝突}
162:                    XDirection := not(XDirection);
163:                    CurrentBaLL^.ThisBaLL^.XDirection := not(XDirection);
164:                  end
165:                eLse
166:                  begin{左右同方向の衝突}
167:                    CurrentBaLL^.ThisBaLL^.XSpeed := XSpeed;
168:                    XSpeed := XSpd;
169:                  end;
170:              end
171:          eLse
172:              begin{相手が右にある}
173:                if (not(OXDir) and (XDirection)) then
174:                  begin{左右反対方向の衝突}
175:                    XDirection := not(XDirection);
176:                    CurrentBaLL^.ThisBaLL^.XDirection := not(XDirection);
177:                  end
178:                eLse
179:                  begin{左右同方向の衝突}
180:                    CurrentBaLL^.ThisBaLL^.XSpeed := XSpeed;
181:                    XSpeed := XSpd;
182:                  end
183:              end
184:          end
185:      end;{if (CurrentBaLL^.ThisBaLL^Id <> Id)...}
186:    CurrentBaLL := CurrentBaLL^.NextBaLL;{次のボール}
187:  end{whiLe}
188: end;
189: {メソッド終わり}
190: {---------------------------------------------------------}
191: {アニメーションと壁衝突処理}
192: procedure BaLL.Bounce(CurrentBaLL : BaLLList);
193: begin
194:   if VisabLe then
195:     begin{ボールを隠す}
196:       BaLL.PutImage;
197:       VisabLe := FaLse;
198:     end;
199: {Y座標計算}
200:   if (YDirection) then
201:     Y := Y + YSpeed
202:    eLse
```

```
203:     Y := Y - YSpeed;
204:
205:   if (Y < (Radius + 1)) then
206:     begin{床に衝突}
207:       YDirection := not(YDirection);
208:       Y := Y + (2 * YSpeed);
209:       Sound(350);
210:       DeLay(10);NoSound;;
211:     end;
212:   if (Y > (Graph.GetMaxY - Radius - 1)) then
213:     begin{天井に衝突}
214:       Ydirection := not(YDirection);
215:       Y := Y - (2 * YSpeed);
216:       Sound(350);
217:       DeLay(10);NoSound;;
218:     end;
219: {X座標計算}
220:   if (XDirection) then
221:     X := X + XSpeed
222:   eLse
223:     X := X -XSpeed;
224:
225:   if (X < (Radius + 1)) then
226:     begin{左壁に衝突}
227:       XDirection := not(XDirection);
228:       X := X + (2 * XSpeed);
229:       Sound(350);
230:       DeLay(10);NoSound;;
231:     end;
232:   if (X > (Graph.GetMaxX - Radius - 1)) then
233:     begin{右壁に衝突}
234:       XDirection := not(XDirection);
235:       X := X - (2 * XSpeed);
236:       Sound(350);
237:       DeLay(10);NoSound;
238:     end;
239:
240:     CoLLisions(CurrentBaLL);{ボール衝突処理}
241: {新しい座標上にボールを描く}
242:   BaLL.PutImage;
243:   VisabLe := TRUE
244: end;
245: {-----------------------------------------------------------}
246: {ボールリストの生成}
247: procedure Create_BaLL_List;
248: var
249:   Num_BaLLs,counter : integer;
250:   ABaLL : BaLLList;
251: begin
```

```
252:   Randomize;
253:   Num_BaLLs := Random(7) + 1;{最大8個}
254: {1個目のボール生成}
255:   New(ABaLL);
256:   ABaLL^.ThisBaLL := New(BPointer,
257:                     Init(Random(Graph.GetMaxX),
258:                          Random(Graph.GetMaxY),
259:                          10,
260:                          Random(10)+10,
261:                          Random(359)+1,
262:                          Random(6)+1,
263:                          Random(10)+1,
264:                          1));
265:   ABaLL^.NextBaLL := niL;
266:   Head := AbaLL;
267:   TaiL := AbaLL;
268: {2個目以上のボール生成}
269:   for counter :=2 to Num_BaLLs do
270:     begin
271:       New(AbaLL);
272:   ABaLL^.ThisBaLL := New(BPointer,
273:                          Init(Random(Graph.GetMaxX),
274:                          Random(Graph.GetMaxY),
275:                          10,
276:                          Random(10)+10,
277:                          Random(359)+1,
278:                          Random(6)+1,
279:                          Random(10)+1,
280:                          counter));
281:       ABaLL^.NextBaLL := niL;
282:       TaiL^.NextBaLL := ABaLL;
283:       TaiL := AbaLL;
284:     end;{for}
285: end;
286: {-------------------------------------------------------------}
287: {アニメーションフレーム1回分}
288: procedure CycLe;
289: var
290:   ABaLL : BaLLList;
291:
292: begin
293:   whiLe not(KeyPressed) do
294:     begin
295:       ABaLL := Head;
296:       repeat
297:         {衝突と移動処理}
298:         ABaLL^.ThisBaLL^.Bounce(Head);
299:       ABaLL := ABaLL^.NextBaLL;
300:       until ABaLL = niL
```

```
301:      end;
302: end;
303: {------------------------------------------------------------}
304: {ボールオブジェクトの消滅}
305: procedure Destroy_BaLL;
306: var
307:   ABaLL : BaLLList;
308:
309: begin
310:   ABaLL := Head;
311:   whiLe(ABaLL^.NextBaLL <> niL) do
312:     begin{イメージ保存ヒープ領域の解放}
313:       ABaLL^.ThisBaLL^.Done;
314:       ABaLL := ABaLL^.NextBaLL;
315:     end;
316: end;
317: {------------------------------------------------------------}
318: {メインプログラム}
319: {------------------------------------------------------------}
320: var
321:   GraphDriver : integer;
322:   GraphMode   : integer;
323:   ErrorCode   : integer;
324:
325: begin
326:   GraphDriver := Detect;
327:   DetectGraph(GraphDriver,GraphMode);
328:   InitGraph(GraphDriver,GraphMode,'');
329:   if GraphResuLt <> GrOK then
330:     begin
331:       WriteLn('グラフィックスドライバがありません。',
332:                                 GraphErrorMsg(GraphDriver));
333:       HaLt(1);
334:     end;
335:
336:   Create_BaLL_List;
337:   CycLe;
338:   Destroy_BaLL;
339:   CLoseGraph;
340: end.
```

では、リスト7-2を見てみましょう。10～15行目がオブジェクトのリスト構造です。ポインタ型が入り乱れているのでちょっとわかりにくいかもしれません。図7-1を見てください。リスト構造の頭と尻尾を指す変数HeadとTailは34行目で宣言されていま

すが、プログラムのずっと後のほうに出てきます。

■図 7-1　BallSequence

リスト構造の連鎖は BallSequence として、あるボールから次のボールを指すポインタ NextBall とそのボールの属性、つまりメソッドを示すポインタ ThisBall でできています。プログラムの 11 行目と 13 行目、10 行目と 14 行目との関係と照らし合わせて見てください。

ボールの個数を増やしたいときには、あらたにボールを作ってそのボールの NextBall で Nil を指し、いままで一番リストの一番最後にあったボールの Nil に接続されている NextBall ポインタでそのボールを指してやればよいわけです。同様にして、ボールを取り除くのも簡単にできるはずです。取り除きたいボールの前後のボールをポインタでつなげばよいのです。

7-2-2■高速アニメーション展開のために

ここまでオブジェクトの設計ができたところで、こんどはアニメーションをどうやるかです。前のリスト 7-1 で紹介した GVRAM のプレーンを切り替えるテクニックも有力ですが、問題は図形が複雑になると座標を移動するたびに描き替えていたのではスピードががた落ちになることです。たとえ丸だけの図形でも、描画のための手続きを呼び出すと、円の中心座標から半径で示される範囲のピクセルを指定された色で光らせるという手順が行われます。

座標移動のたびに図形が変化するのであればそれもやむをえませんが、ここでやるボールのように丸くて色と模様がいつも変わらないものですから、あらかじめボールの図形をデータとして持っていて、それを画面に直接描くべきです。画面に示すべき図形

データをビット情報にしたものをビットマップといいます。このプログラムでは、ボールの輪郭と色と模様をあらかじめ作っておき、それをメモリ中にビットマップとして保存し、画面表示はビットマップを直接 GVRAM に書き込んでいます。

このための手続きとして TURBO Pascal の Graph ユニットに PutImage 手続きが用意されています。この手続きの標準形式は次のようになっています。

```
PutImage(x,y:integer; var BitMap; BitBlt:word)
```

ここで x と y は画面座標 (0 〜639,0 〜 399) です。変数 BitMap に表示すべき図形のビットマップが格納されているメモリ領域です。BitBlt は表示の方式を指定する引数で、Graph ユニットで定義済みの定数によりすでに画面に表示されているグラフィックスに対して次のような効果を出します。

CopyPut	：	前のを消して描く
XORPut	：	重なったところのみ消して描く
OrPut	：	重ねて描く
AndPut	：	重なったところだけ描く
NotPut	：	反転イメージを描く

この指定でアニメーションに利用価値があるのは XORPut です。この指定を使用すれば、画面の同じところに 2 度 PutImage すると、最初はグラフィックスを表示し、次には消すという操作ができます。

さて、手続き PutImage で使うビットマップをを格納するための準備がリスト 7-2 の 7、8 行目です。7 行目で array の添字が 0 から 0 になっているのが奇妙な気がします。実はビットマップを格納するためのメモリ領域の大きさは実際に格納すべきグラフィックスが作られるまで決まりません。ビットマップはヒープ領域に格納されますから、その領域の先頭アドレスだけがわかればよいので、添字の大きさはどうでもよいのです。

PutImage を利用し、ヒープ上にグラフィックスのビットマップを格納してそれを表示するための準備手順は次にようになります。

(1) 表示したいグラフィックスを囲む四角形の左上と右下の対角点の座標を決める。
(2) ImageSize 手続きでその四角形の範囲のビットマップに必要なメモリ量をえる。
(3) GetMem 手続きでヒープ上にそのメモリ量を確保する。
(4) Graph ユニットの各種手続きでグラフィックスを描く。
(5) GetImage でビットマップをメモリ内に取り込む。

こうして作ったビットマップによりアニメーションを行うには次の手順でやります。

(1) 一番最初に画面をクリアしておく。
(2) PutImage でビットマップを XORPut 指定で表示する。
(3) 次にグラフィックスを表示する座標を計算する。
(4) 前の座標に XORPut 表示してグラフィックスを消す。

(5) 新しく計算した座標に XORPut する。

(6) (3) ～ (5) を繰り返す。

7–2–3■さて、プログラミング開始

ここまでの予備知識でリスト 7-2 を読んで行きましょう。1 ～ 35 行目までは解説済みですから、38 行目のボールメソッドの定義からです。まず、41 行目の Init です。オブジェクトを発生するメソッドは必ず Constructor で始めなければなりません。そして、手続き名の Init の前にこれがオブジェクト Ball のメソッドであることを示す Ball をつけます。

この手続きの引数は新たにオブジェクト Ball を生成するために必要な情報を引数にしています。ボールを最初に画面内のどの座標に置くかの位置 InitX と InitY 、ボールの半径 InitRadius 、初期速度 InitSpeed 、初期移動角度 InitAngle 、色 InitColor 、模様 InitPattern 、そしてボール番号 InitId です。46 行目の実数変数 Rspeed はこの手続きの中での実数計算用の一時的な変数です。

49 ～ 52 行目は単なる値の代入です。ずっと先にある手続き、Create_Ball_List でボールの中心の最初の座標を乱数で決めている関係で、場合によってはボールの一部が画面の端からはみ出ることがあります。その対策として 54 ～ 59 行目はボール中心から画面の端までの距離が Radius 未満だったら強制的に Radius にしています。

60 ～ 62 行目で初期速度から X 方向と Y 方向の速度成分を三角関数により計算し、四捨五入で正数値をえています。移動角度から毎回速度成分の計算をやると処理速度が低下するので、この計算はここだけにしています。あとで説明しますが、ボールの衝突をどのように処理するかで 1 回だけの計算では済まない場合もあります。

63 、64 行目でボールの色と模様を決めます。65 、66 行目でそれぞれ X 、Y の増える方向、つまり速度の X または Y 成分が正のときを方向が TRUE とするようにします。こうしておいてから速度の絶対値をとって成分速度をすべて正にします。つまり、速度の方向と大きさとを分離します。

70 ～ 79 行目は上に解説したグラフィックスのビットマップ取り込みです。70 行目で座標 (X,Y) にある半径 Radius の円を囲む四角形の領域に必要なメモリ量をえます。72 行目でそのメモリ量をヒープ上に確保します。そこで、73 行目で色をセットし、74 行目で内部を埋める模様 (パターン) を決めてから 75 行目で楕円を描きます。ボールは円形ですから X 座標方向の半径と Y 方向の半径はともに Radius の楕円を描くことになります。円は楕円の特殊な場合ですから、TURBO Pascal では円のみを描く手続きは用意されていません。

この例では使用するパソコンを日本電気の PC-9801 の 640 × 400 ドット表示モードに限定してありますから、アスペクト比についてはまったく考慮していません。しか

し、パソコンによってはアスペクト比によりX方向とY方向の半径を変えないと円として画面表示されないことがあります。ここでアスペクト比とは、

$$\text{アスペクト比} = \frac{\text{横方向のピクセルピッチ}}{\text{縦方向のピクセルピッチ}}$$

になります。テレビではNTSC規格でアスペクト比は4：3＝1.333に決められています。もしも画面に縦横同じ半径の楕円、つまり円を描くと横が30%ほど長めになってしまうというわけです。このアスペクト比がPC-9801では1になっているので円を描けばそのまま表示も円になります。

もしも、アスペクト比が1以外のモニタ画面に円を出力するのであれば、X方向の半径をXRadius、Y方向半径をYRadiuとして、

```
GetAspectRatio(Xasp,Yasp);
XRadius := InitRadius;
YRadius := Round((Xasp/Yasp)*InitRadius);
```

のような計算により求めておく必要があります。むろん、あとのほうに出てくる各種計算もこのアスペクト比を考慮してやらねばなりません。

77行目のGetImageでビットマップを取り込み、80行目のPutImageのXorPutでいま描いたボールのイメージを消しておきます。こうして画面からボールが消えている状態では表示を現す論理変数VisableをFALSEにセットします。こうしておけばある時点でPutImageによりXorPutしたときにそれが描画か消去かを区別できます。

85行目のメソッドDoneはオブジェクトを消滅させるためのものです。このメソッドはかならずDestructorで始めなければなりません。このメソッドは、呼び出されるとヒープ上に生成されたオブジェクトを消します。この例ではオブジェクトは一度作られるとプログラム終了まで存在していますから、特にDestructorは必要ありません。しかし、たとえばボール同士が衝突すると一方のボールが他方に吸収されるようにしたければ、このメソッドによりそのオブジェクトをヒープから取り去ります。一般的にはbeginとendだけで内容が存在しなくてもこの機能は実現されますが、ここではビットマップもヒープ上に取っているのでそれも解放するようにしています。

91行目のPutImageは画面にボールのビットマップをXORPutするものです。95行目にあるDelayの引数はここでは10msにしています。機種によってはこれではボールの動きが速すぎるかもしれません。その場合は試行錯誤で適当な大きさに設定してください。実はボールの個数によってはアニメーションの速度が変わってしまいます。そこで、生成されたボールの個数によりこのDelayの大きさが変わるようにすればよいのですが、ここではめんどうなので適当な大きさを選びました。

99行目のメソッドはボール同士の衝突を処理します。引数のCurrentBallで指されるボールに他のボールが衝突しているかを調べ、衝突したら反発する処理をしています。

XMine と YMine はボールの座標、XPos と YPos は相手ボールの座標、XSpd と YSpd は相手ボールの X 方向と Y 方向の速度成分、OXDir 、OYDir はその速度成分の方向です。ORad は相手ボールの半径です。

処理は 107 行目からです。110 行目で調べるべき相手ボールをリストの先頭からに設定します。111 行目では自分以外の相手ボールすべてを調べるように while 文を組み立てています。115 ~ 121 行目は相手ボールの各種情報を取り出す操作です。これをやっておかないと、いちいちポインタを使って情報を引き出すことになり、プログラムが見づらくなるのでこうしているのです。

衝突の判定は 123 行目です。ここでは自ボールと相手ボールの中心の X 座標同士と Y 座標同士の差がどちらも両ボールの半径の和より小さいときに衝突したと判定しています。本当は自ボールの中心座標を (x1,y1) 、相手ボールの座標を (x2,y2) としたときに、ボールが衝突したかどうかはその 2 つの点の距離、

$$(x1 - x2)^2 + (y1 - y2)^2$$

が両方のボールの半径の和より小さいときのはずです。しかし、さらによく考えるとこの議論が成り立つのはアスペクト比が 1 のときだけです。それにここまで正確にやらなくても人間の目は速く動いているグラフィックスを正確に捕らえられませんから、不自然でない範囲でできるだけやさしい方法を使ってかまいません。このような「だまし」のテクニックは特に高速のアニメーションをしたいときに必要になります。

衝突の処理は 126 ~ 184 行目です。127 、128 行目で衝突の音を出します。相手ボールと自ボールが反対方向だったら方向を交換し、同方向だったら速度を交換します。これも本当は衝突する角度に応じた複雑な物理計算が必要なのですが、ここも自然に見える範囲で簡単な計算で済ませてあります。186 行目で次のボールを取り出して、111 行目の while 文を繰り返します。

192 行目のメソッド Bounce はボールの移動計算と画面の端に来たときのバウンドの処理です。このメソッドは毎回呼び出されますから、ここにアニメーションのメカを入れ込みます。193 行目でボールが画面表示されていないのであれば、画面表示し表示スイッチである Visable を FALSE にしておきます。

199 行目で Y 方向の向きにより次のボールの Y 座標を計算します。205 ~ 217 行目はその座標の値によりボールが画面の上端か下端、つまり天井か床に衝突したかを調べ、衝突するのなら端から折り返す分を倍にして Y 座標を計算しています。これも正確な計算ではありません。本当は端からボールの半径分離れたところでバウンドするのですから、バウンドしたときの座標もそれにしたがってしなければなりません。同様にして左右の壁に衝突したときにも折り返し分を倍にするだけで済ませています。

座標の計算と画面の端への衝突の処理が終わったら、240 行目で他のボールと衝突しているかどうかの判定と処理を Collisions で行い、新しい座標を決めてから 242 行目で

ボールを画面表示します。表示スイッチの Visable を TRUE にしておきます。

これでメソッドを終わります。この後はボールを作りだし、アニメーションに取りかかります。後はボールオブジェクトと環境の関係を記述することになります。247 行目からはボールリストを生成する手続き、Create_Ball_List です。Num_Balls は生成するボールの個数、counter は手続き内で使用する整数、ABall はボールリストです。

この手続きではボールを乱数で作ります。253 行目で最大 8 個のボールを作るようにしています。1 を加えているのは少なくとも 1 個以上作られるようにするためです。255 〜 267 行目は 1 個目のボールの生成です。255 行目でリストを新たに作り、256 行目でボールオブジェクトを作ります。ここで使用している New は TURBO Pascal 特有の拡張された New です。この New で BPointer で指されるボールオブジェクトをヒープ上に生成します。ここでは Init メソッドにより、ボールの中心座標を乱数で決め、半径を 10 ピクセル、速度を 10 〜 20 ピクセルの乱数、初期移動方向角度を 0 〜 360 度の乱数にします。ボールの色は 0 〜 14 の乱数に 1 を加えることにより 1 〜 15 の乱数として、0 である黒を除外した色が指定されるようにしています。同じように、ボールの模様もフィルパターンとして 12 種類の中からバックグラウンドとフォアグラウンド、ユーザー定義パターンである 0 番、1 番と 12 番を除外して乱数で決めています。

ここまででではまだボールが 1 個目ですから、265 行目で次のボールを指すポインタは NIL、つまり何も指さないようにしています。当然、リストの先頭と最後を指すポインタもこのボールしかないので、266 、267 行目のようにしておきます。

269 〜 284 行目は 2 個目以上のボールの生成です。ただし、新たに生まれたボールはそれ以前に生まれたボールの後ろにつきますから、リストの最後尾にくるようにします。それが 282 、283 行目です。

288 行目からの手続き Cycle はアニメーションのサイクルを制御します。293 行目でキーが押下されるまでアニメーションサイクルである 294 〜 301 行目を実行します。295 行目でポインタをリストの先頭、つまり 1 個目のボールにセットし、296 行目でリストの最後までの繰り返しを行います。298 行目がボールの移動、衝突、画面表示で、299 行目はリストを先に進めます。

305 行目の手続き Destroy_Ball はリスト上を順にたどってオブジェクトの消滅と、ビットマップに確保したメモリの解放を行います。実は、この手続きはなくてもかまいません。このプログラムはボールのアニメーションをやるだけで、それに引き続き何か他のことをやるわけではありませんから、ヒープメモリを解放する必要はないからです。プログラムが終了すると TURBO Pascal がすべてのメモリをシステムに返しますから、メモリを取ったまま終了してもかまいません。むろん、第 3 章で説明した TSR(常駐終了) だけは違います。

さて、これで長々としたメソッドと手続きの解説は終わりです。もっとも、これほど長い準備をしたおかげで、メインプログラムはとても簡単になります。326 〜 334 行目

までは前節で説明したグラフィックスドライバ使用の定石です。336 行目以下はすでに解説した手続きを並べただけです。最後に 339 行目でプログラム終了前にグラフィックスドライバの使用の終了を行います。

以上のプログラムを見ていて、不思議に思いませんでしたか。オブジェクトは var によりインスタンスとして変数宣言されてはじめて使用できるはずですが、それがどこにもなかったのです。実はインスタンスの宣言は 250 行目のリスト構造の変数 ABall の宣言で行われていたことに気付いてください。

どうでしたか。オブジェクトを使ってのプログラミングもなれてしまえば楽なものです。TURBO Pascal のマニュアルのオブジェクト指向プログラミングの最後にも書いてありましたが、「あっ、そうか！」とわかった瞬間にオブジェクト指向プログラミングが自分のものになります。

ここで作ったリスト 7-2 はボールの衝突運動をシミュレーションしたのですが、衝突の判定にしても、衝突の処理にしてもごまかしの計算で済ませました。物理が得意なひとならば、完全弾性衝突にしたり、衝突を繰り返すごとにエネルギーを失って最後はボールがのろのろ動くようにしたり変えられるはずです。さらに、衝突の条件によっては 2 つのボールが 1 個になったり、衝突した相手のボールが 2 つに分裂したりとまるで原子核の運動のようなプログラムに変えてみるのもおもしろいでしょう。あるいは、面の下方に向かって重力があるとすると、またボールの運動が変わってきます。

どう変えてるのも、すでにオブジェクトを自分のものにできたあなたにはそれほど難しいことではないでしょう。ボールオブジェクトの属性を変えたり、オブジェクトの環境を変化させるだけで、このプログラムをいろいろ発展させて遊んでください。

最後になりましたが、このボールアニメーションは、J.G.Soybel "Turbo Pascal 5.5 Programing", Windcrest の例題を参考にしたことをお断りしておきます。

スクリーンに直接アクセスする

7–3■高速アセンブラプログラム

スクリーンにテキストを書き込むには Write や WriteLn があることは基礎中の基礎です。それよりはるかに高速にスクリーンに文字を書き込む機械語プログラムを TURBO Pascal で開発してみましょう。

パソコンが 32 ビット CPU 時代に突入した今となっては、わざわざ機械語でプログラムを作らなくても十分な速度が得られるようになりました。しかし、速くなったらなったで、もっと高速にしたいと思うのが人情 (欲張り?) というものです。ここで紹介する方法は、どうしても高速で処理したいルーチンをどうやって TURBO Pascal に取り込むのというテクニックです。

さて、TURBO Pascal にはプログラム中に直接機械語を書き込むため Inline 文が用意されています。Inline の文の書式は

```
Inline(xx/yy/zz/...);
```

です。xx 、yy などは整数定数、変数名、手続き名、関数名、ロケーション・カウンタ (*で表す) のいずれでもかまいません。

■リスト 7-3

```
 1: program List7_3;
 2:
 3: uses Crt;
 4:
 5:   begin
 6:     CLrScr;
 7:     GoToXY(20,10);
 8:     WriteLn('どのキーでも押せばブザーは止まります。');
 9:     InLine
10:       ($b4/$17/
11:        $cd/$18);
12:
13:     repeat
14:     untiL KeyPressed;
15:
16:     InLine
17:       ($b4/$18/
18:        $cd/$18);
19:     CLrScr;
20:     GoToXY(20,10);
21:     WriteLn('デモを終わります。');
22:   end. { sampLe }
```

リスト 7-3 は Inline 文を利用して、毎度おなじみの BIOS を利用したブザー鳴動と停止です。わずか全体で 4 バイトだけですから、Inline 文による高速化の効果といってもたかがしれています。

MOV 、INT の機械語である 16 進数を 8086 の資料から探してくるのはやっかいです。Inline 文を使えば高速なプログラムが作れることはわかっても、いきなり $8A 、... のように 16 進数で 8086 の機械語を書くのは神業です。私の知っているパソコン・マニアにはこの手の「神業」を平気でやるひとがいくらもいます。私もその昔、コンピュータにまだ真空管が使用されている時代にはそんなことをやっていましたが、さすがに寄る年波 (?) には勝てずというかズルくなって楽をするようになりました。

その方法をここで紹介することにしましょう。さらに Inline 文でどの程度高速化できるか様子をみるために、画面全部を文字で埋め尽くしてみます。リスト 7-4 を見てくだ

さい。このプログラムでは普通の Write 文で画面に数字を記入しています。プログラム自体はとても簡単ですから、何をやっているかすぐにわかると思います。

■リスト 7-4

```
 1: program LIST7_4;
 2:
 3: uses Crt;
 4:
 5:   var
 6:     y : integer ;
 7:
 8:   begin
 9:     CLrScr;
10:     GoToXY(20,10);
11:     Write('普通の書き込みです。');
12:     DeLay(500);
13:
14:     for y := 1 to 25 do
15:       begin
16:         GoToXY(0,y);
17:         Write('01234567890123456789012345678 90',
18:               '123456789012345678901234567890',
19:               '1234567890123456789');
20:       end;
21:
22:     DeLay(500);
23:     CLrScr;
24:     GoToXY(20,10);
25:     Write('デモを終わります。');
26:     DeLay(500);
27:     CLrScr;
28:   end.
```

今度は機械語プログラムの開発です。いきなり 16 進数でプログラムを作る神業をやらず、MS-DOS の付録に付いてくる MASM(マクロアセンブラ) を利用して楽をします。アセンブラでプログラムを開発し、それを MASM でアセンブルして機械語を手に入れようというのです。とはいえ、MASM を使うにはマクロ疑似命令とアセンブラの知識が必要です。

なにやら面倒な名前が出てきましたが、しばらくの間がまんしてください。機械語で高速な画面書き込みを行うには、データをいきなり VRAM に入れてしまうのがよさそうです。そこでこの部分を次の形式の手続きとして作ることにしましょう。

```
procedure fastwrite(col,row,attrib:byte;var str:str80);
```

このfastwriteでは画面上のrow行目のcol番目に属性attribを持った文字列strを表示します。

リスト7-5を見てください。3、4、5行にあるrow、col、attribはそれぞれ、16進数の1111、2222、3333に設定されています。これには別に意味はありません。ただ後ほどアセンブルして機械語が手に入ったときに、どの部分をrow、col、attribに置き換えたらよいのかすぐにわかるための工夫です。この例でもわかるように、値引数はとにかく目立つ値に設定して、プログラムの先頭に置きます。スカラ型の値引数は1ワード、つまり2バイトですからここではそれぞれ、適当な2バイトの変数に設定したのです。

それに対して変数引数はプログラムのセグメント疑似命令のすぐ後ろに置きます。変数引数はサブプログラムへそのベースアドレスとオフセットがそれぞれ1ワードの計2ワードで引き渡されます。これは変数の型によりません。ですから、リスト7-5の10行目でdd(ダブルワード)で2バイト確保します。アセンブラのプログラムを開発するときにはTURBO Pascalのマニュアルの引数の説明をよく読んでください。

7行目と12行目はおまじないとでも思ってこの通りにいつもプログラムを作ってください。むろん、手元にMS-DOSのMASMのマニュアルがあれば、これらがどういうことなのかはすぐにわかります。13行目からアセンブラのプログラムが始まります。15～22行はこれから書き込む座標、つまりテキストVRAMでの位置を計算して、それをDIに入れています。図7-2にテキストVRAMの表示エリアを示します。テキスト画面は2枚あります。

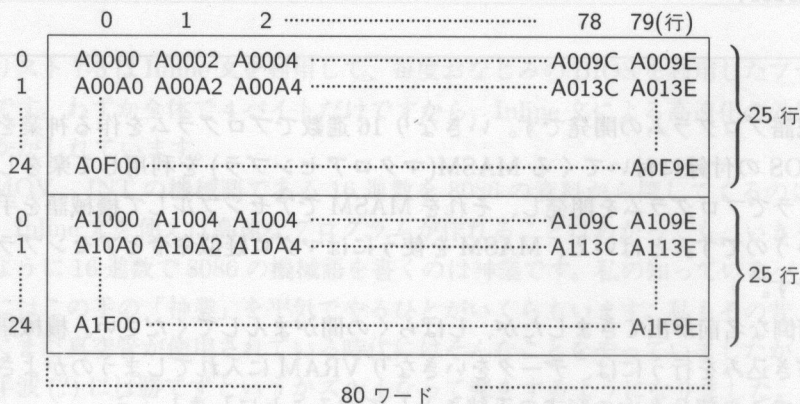
23～30行は文字列を取り出すための準備作業です。TURBO Pascalでは文字列の第1バイトにはその文字列の長さが入っていることを思いだしてください。32～35行でVRAMへの書き込みの準備としてアトリビュート(属性)をセットしています。

■図 7-2 テキストVRAM表示エリア

図 7-3 にアトリビュートのバイト形式を示しておきます。これを指定に従って操作すればかなりおもしろい画面が作れるでしょう。textmem 以下で CL に文字長、BX に文字、DS にテキスト VRAM の表示エリアとアトリビュートエリア (表示エリアと同じ) の先頭アドレス、そして DL にアトリビュートを入れて使用しています。

7	6	5	4	3	2	1	0
G	R	B	VL/BG	UL	RV	BL	ST

■図 7-3 アトリビュートバイト形式

37 ~ 60 行のループで文字列ないの 1 文字ずつを次々に VRAM に書き込んでいます。getchar でメモリから文字を BL に取り込みます。次の testlow は画面表示の開始要求を行っています。これはテキスト制御用の GDC(グラフィック・ディスプレイ・コントローラ) の内部にあるコマンド/パラメータを格納する 16 バイトからなるバッファが空くまで待つためです。こうして、GDC が他のコマンドを実行していないことを確認するのです。

リスト 7-5 のプログラムのおおよそを説明しましたが、アセンブラがよくわからない読者は、こんなプログラムは作れそうもないとあきらめるかも知れません。しかし、ちょっと待ってください。せっかく作ったプログラムですから、どうやってそれを TURBO Pascal に組み込むのか知ってからでも遅くありません。いろいろな本に載っているアセンブラで作られているプログラムも自分で作り変えられるところを手直しすれば、TURBO Pascal への組み込み方さえわかれば十分使いこなせるはずです。

■リスト 7-5

```
 1: ; LIST7_5
 2:
 3: row          equ          1111h
 4: coL          equ          2222h
 5: attrib       equ          3333h
 6:
 7: codeseg      segment      para
 8:              assume       cs:codeseg,es:nothing,ds:nothing
 9:
10: str          dd           0000h
11:
12: fastwrit     proc         far
13:              push         ds
14:              push         ds
15:              mov          aL,row[bp+0]
16:              mov          bL,80
```

```
17:                 muL         bL
18:                 sub         bx,bx
19:                 mov         bL,coL[bp+0]
20:                 add         ax,bx
21:                 add         ax,ax
22:                 mov         di,ax
23:                 mov         bh,00h
24:                 Les         si,str[bp+0]
25:                 sub         cx,cx
26:                 mov         cL,es:[si]
27:                 sub         ax,ax
28:                 pop         ds
29:                 and         cL,cL
30:                 jz          done
31:
32: textmem:        mov         dx,0a000h
33:                 mov         ds,dx
34:                 mov         dh,00h
35:                 mov         dL,attrib[bp+0]
36:
37: getchar:        inc         si
38:                 mov         bL,es:[si]
39:
40: testLow:        in          aL,60h
41:                 test        aL,04h
42:                 jz          testLow
43:                 mov         aL,0dh
44:                 out         62h,aL
45:                 cLi
46:
47: testhi:         in          aL,60h
48:                 test        aL,04h
49:                 jz          testLow
50:                 mov         ds:[di],bx
51:                 push        ds
52:                 push        dx
53:                 mov         dx,0a200h
54:                 mov         ds,dx
55:                 pop         dx
56:                 mov         ds:[di],dx
57:                 pop         ds
58:                 inc         di
59:                 inc         di
60:                 Loop        getchar
61:
62: done:           pop         ds
63:
64: fastwrit        endp
65: codeseg         ends
```

```
66:          end
```

リスト 7-5 のプログラムが完成したとして話を先に進めましょう。このプログラムの作成はむろんターボ・エディタで十分作れます。このプログラムの名前を LIST7-5.PAS にしておきます。MS-DOS の MASM でアセンブルした結果がリスト 7-1 のアセンブル結果です。これでえられた結果の機械語 1E 、1E 、8A 、86 、... を Inline 文の中身にすればよいのです。

その結果できた TURBO Pascal の高速画面書き込み手続き fastwrite をリスト 7-6 に示します。Inline 文の中に現れる変数名はどうしてそこに入るかを、リスト 7-5 のアセンブル結果と比較して調べてみてください。

■リスト 7-6

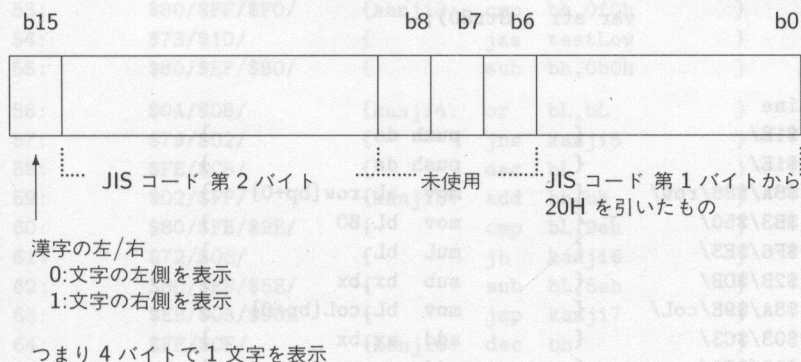
```
 1: {LIST7_6}
 2:
 3: procedure FastWrite(coL,row,attrib : byte ;
 4:                  var str : str80);
 5:
 6:   begin
 7:     inLine
 8:       ($1E/              {          push ds             }
 9:        $1E/              {          push ds             }
10:        $8A/$86/row/      {          mov  aL,row[bp+0]   }
11:        $B3/$50/          {          mov  bL,80          }
12:        $F6/$E3/          {          muL  bL             }
13:        $2B/$DB/          {          sub  bx,bx          }
14:        $8A/$9E/coL/      {          mov  bL,coL[bp+0]   }
15:        $03/$C3/          {          add  ax,bx          }
16:        $03/$C0/          {          add  ax,ax          }
17:        $8B/$F8/          {          mov  di,ax          }
18:        $B7/$00/          {          mov  bh,00h         }
19:        $C4/$B6/str/      {          Les  si,str[bp+0]   }
20:        $2B/$C9/          {          sub  cx,cx          }
21:        $26/$8A/$0C/      {          mov  cL,es:[si]     }
22:        $2B/$C0/          {          sub  ax,ax          }
23:        $1F/              {          pop  ds             }
24:        $22/$C9/          {          and  cL,cL          }
25:        $74/$31/          {          jz   done           }
26:        $BA/$A000/        {textmem:  mov  dx,0a000h      }
27:        $8E/$DA/          {          mov  ds,dx          }
28:        $B6/$00/          {          mov  dh,00h         }
29:        $8A/$96/attrib/   {          mov  dL,attrib[bp+0]}
30:        $46/              {getchar:  inc  si             }
31:        $26/$8A/$1C/      {          mov  bL,es:[si]     }
32:        $E4/$60/          {testLow:  in   aL,60h         }
33:        $A8/$04/          {          test aL,04h         }
```

```
34:        $74/$FA/        {          jz   testLow      }
35:        $B0/$0D/        {          mov  aL,0dh       }
36:        $E6/$62/        {          out  62h,aL       }
37:        $FA/            {          cLi               }
38:        $E4/$60/        {testhi:   in   aL,60h       }
39:        $A8/$04/        {          test aL,04h       }
40:        $74/$FA/        {          jz   testLow      }
41:        $89/$1D/        {          mov  ds:[di],bx   }
42:        $1E/            {          push ds           }
43:        $52/            {          push dx           }
44:        $BA/$A200/      {          mov  dx,0a200h    }
45:        $8E/$DA/        {          mov  ds,dx        }
46:        $5A/            {          pop  dx           }
47:        $89/$15/        {          mov  ds:[di],dx   }
48:        $1F/            {          pop  ds           }
49:        $47/            {          inc  di           }
50:        $47/            {          inc  di           }
51:        $E2/$DA/        {          Loop getchar      }
52:        $1F);           { done:    pop  ds           }
53:   end;
```

リスト 7-7 はリスト 7-4 のプログラムを改造して、fastwrite がどの位速いかをデモさせるものです。Fast screen updating というメッセージに続いて高速画面書き込みが行われますが、ドンという感じで一瞬のうちに画面書き込みが完了します。

■リスト 7-7

```
 1: program LIST7_7;
 2:
 3: uses Crt;
 4:
 5:   type
 6:     Str80 = string[80] ;
 7:
 8:   var
 9:     x : Str80 ;
10:     y : byte ;
11:
12:   {$I \ibmpc\LIST7-6.PAS}
13:
14:   begin
15:     CLrScr;
16:     GoToXY(20,10);
17:     Write('普通の書き込みです。');
18:      DeLay(500);
19:
20:     for y := 0 to 24 do
21:       begin
```

```
22:        GoToXY(0,y);
23:          Write('01234567890123456789012345678901234567890',
24:                '1234567890123456789012345678901234567890',
25:                 '123456789012345678901234567890123456789');
26:       end;
27:
28:     DeLay(500);
29:     CLrScr;
30:
31:     GoToXY(20,10);
32:     Write('高速の書き込みです。');
33:     DeLay(250);
34:
35:     x:='01234567890123456789012345678901234567890'+
36:        '123456789012345678901234567890123456789';
37:
38:     for y:=0 to 24 do
39:       FastWrite(0,y,$e1,x);
40:
41:     DeLay(500);
42:     CLrScr;
43:     GoToXY(20,10);
44:     Write('デモを終わります。');
45:     DeLay(500);
46:     CLrScr;
47:   end.
```

■図 7-4 漢字表現形式

せっかくここまでやったのですから、今度は漢字も表示できるようにさらに fastwrite を改造します。MS-DOS 上で走る関係から、TURBO Pascal の内部では漢字は 2 バイトのシフト JIS コードで表現されています。一方、PC-9801 の内部では 2 バイトの JIS コードで表現されているので、VRAM に書き込む際にシフト JIS-JIS へのコード変換

をしてやらねばなりません。しかも、この2バイトのコードをVRAMの偶数アドレス、奇数アドレスに書き込まねばなりません。

さらに、PC-9801のVRAMに書き込むときにはJISコードそのままではないのです。図7-4にその形式を示しておきます。

これを考慮して手続きfastwriteを改造したのがリスト7-8です。主な改造箇所は40行以下のkanjiで始まるラベルのところです。kanjiからkanji7までがシフトJISをJISコードに変換している部分です。getcharのところで、入力文字が漢字かANK文字かを判断しています。

この判断はシフトJISコードの第1バイトが80H～9FH、E0H～FFHであるので、簡単にできます。入力文字が漢字のときには2バイトで1文字ですから、CLを1減らしています。testhiで文字を書き込むときに、漢字かどうかを判定するため、BHが0でないかどうかを見ています。漢字はBX全体を使用するので、これで判定できるからです。表示文字が漢字だと、テキストVRAM表示エリアを4バイト、アトリビュート・エリアを2バイト使用します。

103行目でループが終了すると105行目にいったんジャンプし、そこからさらに32行目にジャンプする2段構えになっているのは、LOOPや条件付きジャンプでは－126～＋129バイト以内しか分岐できないための工夫です。

■リスト7-8

```
 1: {LIST7_8}
 2:
 3: procedure FastWrite(coL,row,attrib : byte ;
 4:                        var str : Str80);
 5:
 6:   begin
 7:     InLine
 8:      ($1E/              {          push ds          }
 9:       $1E/              {          push ds          }
10:       $8A/$86/row/      {          mov  aL,row[bp+0] }
11:       $B3/$50/          {          mov  bL,80       }
12:       $F6/$E3/          {          muL  bL          }
13:       $2B/$DB/          {          sub  bx,bx       }
14:       $8A/$9E/coL/      {          mov  bL,coL[bp+0] }
15:       $03/$C3/          {          add  ax,bx       }
16:       $03/$C0/          {          add  ax,ax       }
17:       $8B/$F8/          {          mov  di,ax       }
18:       $B7/$00/          {          mov  bh,00h      }
19:       $C4/$B6/str/      {          Les  si,str[bp+0] }
20:       $2B/$C9/          {          sub  cx,cx       }
21:       $26/$8A/$0C/      {          mov  cL,es:[si]  }
22:       $2B/$C0/          {          sub  ax,ax       }
23:       $1F/              {          pop  ds          }
24:       $22/$C9/          {          and  cL,cL       }
```

```
 25:    $74/$03/           {          jz   done1           }
 26:    $EB/$04/$90/       {          jmp  graphics        }
 27:    $E9/$AA/$00/       {done1:    jmp  done            }
 28:    $BA/$A000/         {textmem:  mov  dx,0a000h       }
 29:    $8E/$DA/           {          mov  ds,dx           }
 30:    $B6/$00/           {          mov  dh,00h          }
 31:    $8A/$96/attrib/    {          mov  dL,attrib[bp+0] }
 32:    $46/               {getchar:  inc  si              }
 33:    $26/$8A/$1C/       {          mov  bL,es:[si]      }
 34:    $80/$FB/$80/       {          cmp  bL,80h          }
 35:    $72/$4D/           {          jb   testLow         }
 36:    $80/$FB/$E0/       {          cmp  bL,0e0h         }
 37:    $73/$05/           {          jae  kanji           }
 38:    $80/$FB/$A0/       {          cmp  bL,0a0h         }
 39:    $73/$43/           {          jae  testLow         }
 40:    $FE/$C9/           {kanji:    dec  cL              }
 41:    $46/               {          inc  si              }
 42:    $26/$8A/$3C/       {          mov  bh,es:[si]      }
 43:    $86/$FB/           {          xchg bh,bL           }
 44:    $81/$FB/$2020/     {kanji1:   cmp  bx,2020h        }
 45:    $75/$03/           {          jne  kanji2          }
 46:    $BB/$2121/         {          mov  bx,2121h        }
 47:    $80/$FF/$80/       {kanji2:   cmp  bh,80h          }
 48:    $72/$2D/           {          jb   testLow         }
 49:    $80/$FF/$A0/       {          cmp  bh,0a0h         }
 50:    $73/$06/           {          jae  kanji3          }
 51:    $80/$EF/$70/       {          sub  bh,70h          }
 52:    $EB/$09/$90/       {          jmp  kanji4          }
 53:    $80/$FF/$F0/       {kanji3:   cmp  bh,0f0h         }
 54:    $73/$1D/           {          jae  testLow         }
 55:    $80/$EF/$B0/       {          sub  bh,0b0h         }
 56:    $0A/$DB/           {kanji4:   or   bL,bL           }
 57:    $79/$02/           {          jns  kanji5          }
 58:    $FE/$CB/           {          dec  bL              }
 59:    $02/$FF/           {kanji5:   add  bh,bh           }
 60:    $80/$FB/$9E/       {          cmp  bL,9eh          }
 61:    $72/$06/           {          jb   kanji6          }
 62:    $80/$EB/$5E/       {          sub  bL,5eh          }
 63:    $EB/$03/$90/       {          jmp  kanji7          }
 64:    $FE/$CF/           {kanji6:   dec  bh              }
 65:    $80/$EB/$1F/       {kanji7:   sub  bL,1fh          }
 66:    $86/$FB/           {          xchg bh,bL           }
 67:    $E4/$60/           {testLow:  in   aL,60h          }
 68:    $A8/$04/           {          test aL,04h          }
 69:    $74/$FA/           {          jz   testLow         }
 70:    $B0/$0D/           {          mov  aL,0dh          }
 71:    $E6/$62/           {          out  62h,aL          }
 72:    $FA/               {          cLi                  }
 73:    $E4/$60/           {testhi:   in   aL,60h          }
```

```
 74:        $A8/$04/            {          test aL,04h         }
 75:        $74/$EF/            {          jz   testLow        }
 76:        $80/$FF/$00/        {          cmp  bh,00h         }
 77:        $74/$03/            {          je   ank            }
 78:        $80/$EB/$20/        {          sub  bL,20h         }
 79:        $89/$1D/            {ank:      mov  ds:[di],bx     }
 80:        $80/$FF/$00/        {          cmp  bh,00h         }
 81:        $74/$09/            {          je   ank1           }
 82:        $47/                {          inc  di             }
 83:        $47/                {          inc  di             }
 84:        $80/$C7/$80/        {          add  bh,80h         }
 85:        $89/$1D/            {          mov  ds:[di],bx     }
 86:        $4F/                {          dec  di             }
 87:        $4F/                {          dec  di             }
 88:        $1E/                {ank1:     push ds             }
 89:        $52/                {          push dx             }
 90:        $BA/$A200/          {          mov  dx,0a200h      }
 91:        $8E/$DA/            {          mov  ds,dx          }
 92:        $5A/                {          pop  dx             }
 93:        $89/$15/            {          mov  ds:[di],dx     }
 94:        $80/$FF/$00/        {          cmp  bh,00h         }
 95:        $74/$04/            {          je   ank2           }
 96:        $47/                {          inc  di             }
 97:        $47/                {          inc  di             }
 98:        $89/$15/            {          mov  ds:[di],dx     }
 99:        $1F/                {ank2:     pop  ds             }
100:        $47/                {          inc  di             }
101:        $47/                {          inc  di             }
102:        $B7/$00/            {          mov  bh,00h         }
103:        $E2/$03/            {          Loop getchar1       }
104:        $EB/$04/$90/        {          jmp  done           }
105:        $E9/$61/$FF/        {getchar1:jmp  getchar         }
106:        $1F);               {done:     pop  ds             }
107:   end;
```

リスト 7-9 はリスト 7-8 の漢字対応版の fastwrite を使用した画面高速書き込みのデモプログラムです。漢字コードの変換が入っているにもかかわらず、猛烈な速度で一瞬のうちに画面全体に文字が書き込まれます。

■リスト 7-9

```
  1: program LIST7_9;
  2:
  3: uses Crt;
  4:
  5:   type
  6:     Str80 = string[80] ;
  7:
```

```
 8:  var
 9:    x1 : Str80 ;
10:    x2 : Str80 ;
11:    y  : byte  ;
12:
13:  {$I LIST7-8.PAS}
14:
15:  begin
16:    CLrScr;
17:    GoToXY(20,10);
18:    Write('普通の書き込みです。');
19:    DeLay(1000);
20:
21:    for y := 1 to  12 do
22:      begin
23:        GoToXY(0,2*y);
24:        Write('Our policy is to always blame the computer!',
25:              'コンピュータに責任をとらせましょう。');
26:        GoToXY(0,2*y+1);
27:        Write('GARBAGE IN.....GARBAGE OUT.....GIGO!',
28:              'ゴミが入ればゴミが出る!?');
29:      end;
30:
31:    DeLay(1000);
32:    CLrScr;
33:
34:    GoToXY(20,10);
35:    Write('高速書き込みです。');
36:    DeLay(1000);
37:
38:     x1 :='Our policy is to always blame the computer!'+
39:              'コンピュータに責任をとらせましょう。';
40:     x2 :='GARBAGE IN.....GARBAGE OUT.....GIGO!'+
41:              'ゴミが入ればゴミが出る!?';
42:
43:    for y:=0 to 12 do
44:      begin
45:        FastWrite(0,2*y,$a1,x1);
46:        FastWrite(0,2*y+1,$81,x2);
47:      end;
48:
49:    DeLay(1000);
50:    CLrScr;
51:    GoToXY(20,10);
52:    Write('デモを終わります。');
53:    DeLay(1000);
54:    CLrScr;
55:  end.
```

ここではアセンブラがある程度わかっている読者を対象としましたが、fastwrite は画面に大量に文字を表示するときに利用できるはずです。アセンブラがわからないひとも、おもしろいことができそうなアセンブラのプログラムを雑誌や本で見つけたら、ここで紹介した方法を使って TURBO Pascal の手続きに直して実験してみるとよいでしょう。ただし、そのときはいつ暴走してファイルを壊されてもよいように、バックアップ用のフロッピーで実験してください。

ウィンドウプログラミング

7–4■TURBO Vison をフル活用

私たち素人が作るプログラムとプロが作った売りものになるプログラムの違いはなんでしょうか。プロが作ったプログラムは内容的にもよく整理されていて、後からの保守がやりやすいとか、バグが出にくいようになっているとか、スピードが速いとかが挙げられるでしょう。しかし、最大の違いはユーザーインタフェースの分かりやすさです。つまり、必ずしもパソコンの上級者ではないひとにでも使いやすくできていることです。使いやすいプログラムとは画面を見れば、何をすればよいのかが一目でわかるプログラムということになります。そのために欠かせないテクニックがウィンドウです。

画面に次々にウィンドウを開き、それを拡大、移動したり、そして用が済んだらクローズ (閉じる) したりというプログラムを作るのは大変ですが、ユーザーにとってはとてもありがたいものです。とはいえ、そんなプログラムをどうしたら作れるのか途方にくれてしまいます。

ところが、TURBO Pascal の ver6.0 から追加された TURBO Vision(ターボ・ビジョン) を利用すると、いとも簡単に実現できるのです。TURBO Pascal プログラマーとしてこれを見逃す手はありません。なにしろ、TURBO Pascal の統合環境も TURBO Vision を使って作られているのですから、まずほとんどのウィンドウ環境ならこれでなんとかなりそうです。TURBO Vision 自体の解説は TURBO Pascal に付属してる TURBO Vision ガイドにそのすべてが書いてあります。このガイドはなんと 600 ページ弱という超大作 (?) なので、読むのに少々骨が折れます。頭から読んで行こうとすると途中で挫折しかねません。いきなりこれに取りかかるよりも、オブジェクト指向プログラミングに慣れてからのほうがずっと簡単に理解できます。

TURBO Vision を理解するときに、これはウィンドウ環境を作るためのユニットを集めたライブラリのようなものでないかという先入観があると、何が書いてあるのかわからなくなるのです。しかし、TURBO Vision とはオブジェクトの集合と考えると、すべてが突然すっきりと理解できるようになります。TURBO Vision はウィンドウ環境を作る上で必要となるオブジェクトを提供してくれます。だからといって、TURBO Vision のオブジェクトしか使えないということではないのです。いわば、TURBO Vision はオ

ブジェクトの骨組みを与えるので、こうしたいという目的に沿ったメソッドを書けば、提供されているオブジェクトのメソッドをオーバーライトすることができます。

しかも、オーバーライトした部分以外は上位のオブジェクトのメソッドが継承されるので、変更は必要最小限で済みます。もしもTURBO Visionが単なるライブラリ集であったら、変更するときには変更する部分を含むプログラムのソースコードを書き換えなければなりません。こんなことをするとバグが生まれるもとになります。すでにデバッグが終わって完全に動作することが確認されているオブジェクトのメソッドには何も手を加えず、変更部分だけに集中してプログラムを作れるのがオブジェクト指向を全面的に取り入れたTURBO Visionのすばらしいところです。

TURBO Visionの機能は非常に多いので、ここですべてを解説することはとても無理です。ここではTURBO Visionの使い方の実例をながめて、どんなものかを味見する程度にしておきます。リスト7-10を見てください。これはファイルの内容を読み込んで、画面に開いたウィンドウに次々に表示するものです。プログラムの解説に入る前に、コンパイルして実行してみてください。ただし、このプログラムは統合環境から実行させようとすると、メモリ不足で動きません。TURBO Pascalに付属しているコマンドラインコンパイラTPCでコンパイルしてください。そのときには、3行目のusesで宣言されている各ユニットがつけ加えられるように、ディレクトリを指定しておく必要があります。Dos、Objectsユニットは¥TPに、Drivers、Views、menus、Dialog、Appユニットは¥TP¥TVISIONに、StdDlgとFviewerは¥TP¥TVDEMOSにあります。したがってハードディスクAドライブにTURBO Pascalがインストールされた標準状態で、Bドライブのルートディレクトリ上にあるリスト7-10をコンパイルするのであれば、

```
A>¥TP¥TPC B:¥LIST7-1 /UA:¥TP;¥TP¥TVISION;¥TP¥TDEMOS
```

とします。/記号の後ろのUはこれに続くディレクトリ内からユニットを探してくるようにとの指定です。この例ではカレントドライブがAドライブですから、A：はなくてもかまいません。あなたのTURBO Pascalのインストール状態によりカレントドライブ以外のドライブに上記のユニットが存在するときには適当に変えてください。

さて、コンパイルして実行してみた結果はどうだったでしょうか。これほど複雑な動きをするプログラムがわずか120行程度で作れたことに驚いたでしょう。こんなプログラムならマウスの使いでもあります。むろん、マウスのない場合でもTABキー、矢印キーなどを組み合わせれば動作します。

■リスト7-10

```
  1: program LIST7_10;
  2:
```

```
 3: uses
 4:   Dos, Objects, Drivers, Views, Menus, Dialogs,
 5:    StdDlg, App, FViewer;
 6:
 7: const
 8:   cmFileOpen    =    100;
 9: type
10:
11:   TV_SamplePtr = ^TV_Sample;
12:
13:   TV_Sample = object(TApplication)
14:     constructor Init;
15:     procedure FileOpen(WildCard: PathStr);
16:     procedure HandleEvent(var Event: TEvent); virtual;
17:     procedure InitMenuBar; virtual;
18:     procedure InitStatusLine; virtual;
19:     procedure ViewFile(FileName: PathStr);
20:   end;
21:
22: constructor TV_Sample.Init;
23: begin
24:   TApplication.Init;
25:   RegisterObjects;
26:   RegisterViews;
27:   RegisterMenus;
28:   RegisterDialogs;
29:   RegisterApp;
30:   RegisterFViewer;
31: end;
32:
33: procedure TV_Sample.FileOpen(WildCard: PathStr);
34: var
35:   Direc: PFileDialog;
36:   FileName: PathStr;
37: begin
38:   Direc := New(PFileDialog, Init(WildCard, 'Select File',
39:     '~N~ame', fdOpenButton, 100));
40:   if ValidView(Direc) <> nil then
41:   begin
42:     if Desktop^.ExecView(Direc) <> cmCancel then
43:     begin
44:       Direc^.GetFileName(FileName);
45:       ViewFile(FileName);
46:     end;
47:     Dispose(Direc, Done);
48:   end;
49: end;
50:
51: procedure TV_Sample.HandleEvent(var Event: TEvent);
```

```
 52: begin
 53:   TApplication.HandleEvent(Event);
 54:   if Event.What = evCommand
 55:    then
 56:     if Event.Command = cmFileOpen
 57:      then
 58:       begin
 59:         FileOpen('*.*');
 60:         ClearEvent(Event);
 61:       end
 62:      else
 63:       Exit;
 64: end;
 65:
 66: procedure TV_Sample.InitMenuBar;
 67: var
 68:   R: TRect;
 69: begin
 70:   GetExtent(R);
 71:   R.B.Y := R.A.Y+1;
 72:   MenuBar := New(PMenuBar, Init(R, NewMenu(
 73:     NewSubMenu('~F~ile', hcNoContext, NewMenu(
 74:       NewItem('~O~pen...', 'F3', kbF3, cmFileOpen, hcNoContext,
 75:       NewLine(
 76:       NewItem('E~x~it', 'Grph-X', kbAltX, cmQuit, hcNoContext, nil)))),
 77:     NewSubMenu('~W~indow', hcNoContext, NewMenu(
 78:       NewItem('~Z~oom', 'F5', kbF5, cmZoom, hcNoContext,
 79:       NewItem('~N~ext', 'F6', kbF6, cmNext, hcNoContext,
 80:       NewItem('~C~lose', 'Grph-F3', kbAltF3, cmClose, hcNoContext, nil))),
 81:       nil)))));
 82: end;
 83:
 84: procedure TV_Sample.InitStatusLine;
 85: var
 86:   R: TRect;
 87: begin
 88:   GetExtent(R);
 89:   R.A.Y := R.B.Y - 1;
 90:   StatusLine := New(PStatusLine, Init(R,
 91:     NewStatusDef(0, $FFFF,
 92:       NewStatusKey('~Grph-X~ Exit', kbAltX, cmQuit,
 93:       NewStatusKey('~F3~ Open', kbF3, cmFileOpen,
 94:       NewStatusKey('~Grph-F3~ Close', kbAltF3, cmClose,
 95:       NewStatusKey('~F5~ Zoom', kbF5, cmZoom, nil)))), nil)));
 96: end;
 97:
 98: procedure TV_Sample.ViewFile(FileName: PathStr);
 99: var
100:   W: PWindow;
```

```
101: begin
102:   W := New(PFileWindow,Init(FileName));
103:
104:   if ValidView(W) <> nil then
105:     Desktop^.Insert(W);
106: end;
107:
108: var
109:   Sample: TV_Sample;
110:
111: begin
112:   Sample.Init;
113:   Sample.Run;
114:   Sample.Done;
115: end .
```

それではプログラムの中身を見てみましょう。プログラムを解読するにあたっては、TURBO Visionガイドと実際に各種ユニットを画面に表示してください。先ほどの3行目のuses宣言されている各ユニットの内容の説明はマニュアルにゆずるとして、11行目から始まるオブジェクトTV_Sampleの中身を見てみましょう。11行目のオブジェクトのポインタ化はもうすでに7-2節でおなじみです。13行目でTV_SampleがTApplicationの下位のオブジェクトになっています。

TURBO Visionを使用するアプリケーションは必ずTApplicationの下位オブジェクトとして定義されます。ここではそれをTURBO Visionのサンプルということで、TV_Sampleと名付けておきます。プログラムの終わりのほうにあるメインプログラムではTV_SampleのインスタンスをSampleとして、わずか3行で初期化、ユーザーとの対話、そして終了処理を行っています。これがTURBO Visionのプログラムのいわば定形になります。

14～19行目はTApplicationのメソッドをオーバーライトしているメソッドの宣言です。それぞれの具体的な内容は22行目から始まります。最初は初期化です。22行目で画面の上端にあるメニューバー、下端にあるステータスライン、デスクトップと呼ばれる画面を初期化します。25～30行目は各種オブジェクトを汎用入出力(ストリーム)可能とするための登録です。

33行目はメニューのFileの中からOpenを選んだときに、画面に何を表示してそこから何を手に入れるかのメソッドです。35行目のDirecは¥TVISIONの中にあるユニットStdDlgで定義されているオブジェクトPfileDialogのインスタンスです。このオブジェクトは画面にウィンドウを開き、かつその中に指定されたパスのファイルの表示、およびクリックボタンの表示をします。36行目はDosユニットのパス名を与える型の変数としてFileNameを宣言しています。

38行目のNewによりオブジェクトを生成し、40～46行目でウィンドウ内のクリッ

クボタンがCancelでない場合、つまりなんらかのファイルが選択されたときの処理をします。ここではカレントディレクトリ上のファイルのみの表示と選択を行っていますが、ユニットStdDlgは開いたウィンドウ内でディレクトリの変更もできるよう作られています。キャンセルされるか、なにかファイルが選ばれればこのオブジェクトは存在する必要がありませんから、47行目で破壊されます。

51行目からはイベント、つまりファイルが選ばれたときの処理です。イベントのコマンドとしてファイルの読み込みが選ばれたときには、59行目のようにすべてのファイルを意味するワイルドカード*.*によりファイルを読み込みます。それ以外のイベントは存在しませんから、63行目で何もしないで外へ出るようにしています。

66行目からはメニューバーを作るメソッドです。68行目は画面内の座標用のレコードです。70行目でアプリケーションの領域を確保して、最上段から1行下までをメニューとします。72行目はメニューバーを作るときの定石です。実際にこのプログラムを実行したときの画面の様子と比較すればよくわかるでしょう。ここで~印はそれで挟んだ文字をハイライトにします。cmはそのメニュー項目が選ばれたときのコマンド、hcはそのメニュー項目に対するヘルプを示します。ここではヘルプ機能は付けていませんから、hcNoContextとして何もヘルプがないようにしています。

84行目からは画面の最下段に表示されるステータスラインを作るメソッドです。89行目でステータスラインを画面の最下段から1行目に設定します。中身の設定法はメニューバーとよく似ています。98行目からは実際に画面にファイル内容をウィンドウ内に表示するためのメソッドです。100行目でPWindowというユニットFViewerで定義されているオブジェクトのインスタンスによってウィンドウ表示をしています。105行目でウィンドウWをデスクトップにインサートすることにより、このウィンドウが自動的に画面表示されます。

ここまで駆け足で解説しましたが、これだけの説明ではTURBO Visionの使用法のあらまししか理解できなかったかもしれません。TURBO Visionの解説だけで本が何冊も書けるほどですから無理もないことです。TURBO Visionは非常に便利ですから、ぜひあなたのプログラミング技術に取り入れたいものです。そのための勉強としては次のステップをお勧めしておきます。

(1)「TURBO Pascalトレーニングマニュアル」のオブジェクト指向プログラミングの項を読む。

(2) 本書の7-2節のオブジェクトによるアニメーションを読む。

(3) TURBO Pascal添付のTURBO VisionガイドをTURBO Pascalのディレクトリ¥TP¥DOCDEMOSに入っている例題TVGUID01 ~ 16.PASを実行しながら読む。

(4) ディレクトリ¥TP¥TVDEMOSに入っている例題を解読する。各例題で使われているオブジェクトを記述したユニットも併せて解読する。

以上の順で勉強すれば、かならずTURBO Visionが使いこなせるようになるはずです。TURBO Pascalがオブジェクト指向プログラミングに拡張されたのはver5.5からですが、TURBO Visionはそれを利用したツールになっています。オブジェクトユニットとして提供されるツールはユーザーに対してソースコードを提供せずに、かつユーザー側の自由度を制限しないために、ソフトハウスにもユーザーにも歓迎すべきコンセプトです。ここのところC言語に押され気味のTURBO Pascalですが、このオブジェクト指向によりプロのソフト開発にも十分使用できる機能を備えました。あなたもぜひTURBO Visionを使いこなし、すばらしいソフトを作ってみてください。

テキスト画面切り替え

7–5■VRAMのBIOSをコントロールする

アプリケーション・プログラムを作っていて、入力によって2つの画面を互い違いに表示しなければならないことが結構あります。そんなときに画面をいちいち書き換えていたのでは速度が遅くなってしまいます。グラフィック画面の切り替えについてはすでに7-1節でトレーニングしましたから、今度は、PC-9801に用意されている2枚のテキスト画面を利用して、それを切り替えて表示するサンプル・プログラムを開発してみましょう。

プログラムの内容は、単にDosユニットのTextBank手続きで第0画面と第1画面を切り換えているだけですから、説明の必要はないでしょう。

■リスト7-11

```
 1: program LIST7_11;
 2:
 3: Uses Crt,Dos;
 4:
 5:   begin
 6:     CLrScr;
 7:     WriteLn(' ***** テキスト画面のデモ ***** ');
 8:     WriteLn(' どのキーでも押せばデモを終わります。 ');
 9:     WriteLn;
10:     WriteLn(' <<テキスト画面1>> ');
11:
12:     TextBank(1);
13:     WriteLn(' ***** テキスト画面のデモ ***** ');
14:     WriteLn;
15:     WriteLn(' <<テキスト画面2>> ');
16:
17:     repeat
18:       TextBank(0);
19:       DeLay(500);
```

```
20:          TextBank(1);
21:            DeLay(500);
22:       untiL KeyPressed;
23:
24:         CLrScr;
25:         TextBank(0);
26:         CLrScr;
27:         WriteLn(' ***** テキスト画面切り替えデモ終り ***** ');
28:     end.
```

テキスト画面2ページ目の使い方として、たとえばディレクトリを記入すれば、画面を切り替えるだけでディレクトリが見ることができます。こうすれば、ファイルを使うアプリケーションなどで、いちいちディレクトリ取得のための操作をしないで済みますから、プログラムがすっきりしますしスピードもアップします。実に簡単なテクニックですが、これ以外にもおもしろい使い方がいろいろ考えられると思います。

カーソル制御

7–6■GDCをコントロールする

カーソルはどこへテキスト入力が行われるのかをユーザーに示すものですが、形は四角いのが普通です。しかし、入力を強く促すときには高速で点滅し、またあるときは形を変えたり、まったく表示しなかったりといろいろ操作したい場合があります。カーソルを操作することなど不可能に思えますが、複雑なことをしないのならPC-9801のμPD7220GDC(グラフィック・ディスプレイ・コントローラ)をTURBO Pascalから操作すれば簡単にできます。

具体的には定義済み配列portでGDCを指定し、それに対してデータを入力してやります。これもサンプル・プログラムを見たほうがすぐにわかると思います。

■リスト 7-12

```
 1: program LIST7_12;
 2:
 3: Uses Crt;
 4:
 5: procedure Standard_Cursor;
 6:
 7:     begin
 8:         Port[$0062] := $4b;
 9:         Port[$0060] := $8f;
10:         Port[$0060] := $0f;
11:         Port[$0060] := $fb;
12:     end;
```

```
13:
14: procedure Fast_Cursor;
15:
16:   begin
17:     Port[$0062] := $4b;
18:     Port[$0060] := $8f;
19:     Port[$0060] := $0f;
20:     Port[$0060] := $f1;
21:   end;
22:
23: procedure SLow_Cursor;
24:
25:   begin
26:     Port[$0062] := $4b;
27:     Port[$0060] := $8f;
28:     Port[$0060] := $0f;
29:     Port[$0060] := $ff;
30:   end;
31:
32: procedure Stop_Cursor;
33:
34:   begin
35:     Port[$0062] := $4b;
36:     Port[$0060] := $8f;
37:     Port[$0060] := $2f;
38:     Port[$0060] := $f3;
39:   end;
40:
41: procedure No_Cursor;
42:
43:   begin
44:     Port[$0062] := $4b;
45:     Port[$0060] := $0f;
46:     Port[$0060] := $0f;
47:     Port[$0060] := $fb;
48:   end;
49:
50: procedure Shape_Cursor;
51:
52:   begin
53:     Port[$0062] := $4b;
54:     Port[$0060] := $8f;
55:     Port[$0060] := $4f;
56:     Port[$0060] := $03;
57:   end;
58:
59: {メインプログラム}
60:   begin
61:     CLrScr;
```

```
62:        writeLn('<<カーソル形状デモ：'+
63:                        'リターンキー押下で先に進みます。>>');
64:        writeLn;
65:        Write('           通常のカーソル ：');
66:        ReadLn;
67:        Fast_Cursor;
68:        Write('           高速点滅       ：');
69:        ReadLn;
70:        SLow_Cursor;
71:        Write('           低速点滅       ：');
72:        ReadLn;
73:        Stop_Cursor;
74:        Write('           点滅停止       ：');
75:        ReadLn;
76:        No_Cursor;
77:        Write('           カーソル消去   ：');
78:        ReadLn;
79:        Shape_Cursor;
80:        Write('           変わりカーソル ：');
81:        ReadLn;
82:        Standard_Cursor;
83:        WriteLn(' <<カーソル形状デモを終わります。>> ');
84:     end.
```

	DB7	DB6	DB5	DB4	DB3	DB2	DB1	DB0
C	0	1	0	0	1	0	1	1
P1	CS	0	0	←		L/R		→
P2	←	BLL →	BD	←		CST		→
P3	←		CF1		→	←	BLH	→

■表 7-1　CSRFORM コマンド

リスト7-12 を見てください。57 行まではカーソルを操作する手続きですが、どれもport に対する代入文だけでできています。ポート番号は$62 と 60 ですが、これは GDC に対する CSRFORM(カーソル・フォーム) 用のコマンド・ライトとパラメータ設定です。その内容は表 7-1 を参考にしてください。

表の中のコマンド内容は次のようになります。

CS：文字表示時のカーソル表示の有無を定義し、0 だと表示なし、1 だと表示します。

L/R：1 行中の表示ライン数を定義し、1(00H) ～ 32(1FH) ラインまでです。

BL：文字表示のカーソル点滅周期と、文字/グラフィックス混在モード時の文字属性用

タイミング信号の点滅周期を定義します。後者の場合は4(01H) ~ 124(1FH) 画面まで4ステップごとに定義します。

BD：文字表示時のカーソル表示のブリンキング(点滅)の有無を定義し、0だとブリンキングし、1でなしとなります。

CST：文字表示時のカーソル表示開始ラインの値を定義し、1(00H) ~ 32(1FH) ラインまでです。

CFI：文字表示時のカーソル表示終了ラインの値を定義し、1(00H) ~ 32(1FH) ラインまでです。

以上の情報から作ったのがリスト7-12の中にある6つの手続きです。メインプログラムは60 ~ 84行です。内容はカーソル操作の手続きを次々に呼び出し、readln文でリターンが入力されるまで変化したカーソルを表示し続けます。

同じようなことはBIOSコールでも行うことができます。形式は、

```
Intr($18,dos_reg);
```

で、dos_regの中のAXレジスタに1001Hを設定すればブリンキング停止、1000Hなら開始します。さらに、11xxH(xxは任意)ならカーソル表示、12xxHなら表示停止、13xxHにDXにテキストVRAMアドレスを設定するとカーソル位置をそこにします。手軽なカーソル操作ならこちらのBIOSコールのほうが便利かも知れません。

ランダム・ボックス

7–7■エスケープシーケンスを使う

画面をいろいろ操作するトレーニングを続けてきましたが、最後は簡易グラフィックスともいうべきグラフィック文字の表示をやってみましょう。TURBO Pascalに付属のGRXや7-1で開発したグラフィック・ルーチンを使用すればたいがいのことはこなせるはずですが、ちょっとメニュー画面を枠で囲いたいようなときにはこれでは大げさです。そこで、PC-9801に用意されているグラフィック文字を利用します。

グラフィック文字は表7-2のように、カタカナも含め80H以降に設定されています。しかし、これをいきなり画面に出力してもシフトJISコードの漢字が表示されるだけでうまく行きません。それをやるためにはMS-DOSのエスケープ・シーケンスを利用します。エスケープ・シーケンスはCRT画面の制御を行う特殊文字列で、MS-DOSでは表7-3のように決めています。

■リスト7-13

```
 1: program LIST7_13;
 2:
```

```
 3: uses Crt;
 4:
 5: const
 6:   ESC = #$1B;
 7:
 8: var
 9:   Lst : Text;
10:
11: procedure BoxColor(coL:byte) ;
12:   var
13:    c : String[3] ;
14:    begin
15:     case coL of
16:       0 : c := '17m'  ;
17:       1 : c := '18m'  ;
18:       2 : c := '19m'  ;
19:       3 : c := '20m'  ;
20:       4 : c := '21m'  ;
21:       5 : c := '22m'  ;
22:       6 : c := '23m'  ;
23:       7 : c := '30m'  ;
24:     end ;
25:     Write(Lst,ESC,'[',c) ;
26:    end ;
27: {----------------------------------------------------}
28:
29: procedure Box(x1,y1,x2,y2 : integer) ;
30:   var
31:    i : integer ;
32:   begin
33:    GoToXY(x1,y1) ;
34:    Write(Lst,#$98) ;
35:    for i := x1+1 to x2-1 do Write(Lst,#$95) ;
36:    Write(Lst,#$99) ;
37:    for i := y1+1 to y2-1 do
38:     begin
39:      GoToXY(x1,i) ;
40:      Write(Lst,#$96)  ;
41:      GoToXY(x2,i) ;
42:      Write(Lst,#$96)  ;
43:     end ;
44:    GoToXY(x1,y2) ;
45:    Write(Lst,#$9A) ;
46:    for i := x1+1 to x2-1 do Write(Lst,#$95) ;
47:    Write(Lst,#$9B) ;
48:   end ;
49: {====================================================}
50: var
51:  r : integer ;
```

```
52:
53: begin
54:   CLrScr ;
55:   Assign(Lst,'CON');
56:   ReWrite(Lst);
57:   Write(Lst,ESC,')3');
58:   Randomize;
59:     for r := 1 to 10 do
60:       begin
61:         BoxColor(Random(7)) ;
62:         Box(1+Random(40),1+Random(12),
63:             40+Random(40),13+Random(11)) ;
64:       end ;
65:   Write(Lst,ESC,')0',ESC,'[23m');
66:   CLrScr;
67: end .
```

リスト 7-13 は画面にランダムな色と大きさの四角を次々に描くプログラムです。このプログラム自体はエスケープ・シーケンスを使っているだけで、実用性は特にありません。後の第 8 章でメニュー・ボックスを描くのに使える程度です。それよりおもしろい使い方があります。このプログラムを COM ファイルとして作って友人に上げるプログラムのコピーディスクのオートバッチファイルに入れておくのです。そのフロッピーをシステム立ち上げに使うと、いきなり次々にカラフルな四角形が画面に現れるので、友人を驚かすにはちょうどよいでしょう。

さて、リスト 7-13 に戻ってみます。11 行目の手続き、BoxColor はテキスト画面の表示色を設定するものです。引数として色番号をもらい、それを MS-DOS のエスケープ・シーケンスに直しているだけです。29 行目の手続き、Box は画面上で対角線になる角の座標を指定して四角形をグラフ文字により描画するものです。#$の後に続く数値はキャラクタ・コード表から探してきた番号です。

メインプログラムは 50 行目からです。57 行目でグラフモード設定用のエスケープ・シーケンスを出力し、これ以後はシフト JIS コードの漢字出力を抑制し、グラフィック文字を表示するようになります。59 ～ 64 行の for 文でランダムな描画をおこないます。61 行目で random の引数を 7 としたのは、8 の黒色の箱を描いても見えないので、表示から外すための配慮です。62 行目では、テキスト画面が 25 字× 40 行であることを考えて作っています。65 行目は本来の日本語 MS-DOS の表示画面である漢字モードに画面を復帰するプログラムです。こうしないと、この後で実行するプログラムの漢字表示がおかしくなります。

	0	1	2	3	4	5	6	7	8	9	A	B	C	D	E	F
0		DE	SP	0	@	P		p			SP	ー	タ	ミ		
1	SH	D1	!	1	A	Q	a	q			。	ア	チ	ム		円
2	SX	D2	”	2	B	R	b	r			「	イ	ツ	メ		年
3	EX	D3	#	3	C	S	c	s			」	ウ	テ	モ		月
4	ET	D4	$	4	D	T	d	t			、	エ	ト	ヤ		日
5	EQ	NK	%	5	E	U	e	u			・	オ	ナ	ユ		時
6	AK	SN	&	6	F	V	f	v			ヲ	カ	ニ	ヨ		分
7	BL	EB	’	7	G	W	g	w			ァ	キ	ヌ	ラ		秒
8	BS	CN	(	8	H	X	h	x			ィ	ク	ネ	リ	♠	
9	HT	EM	)	9	I	Y	i	y			ゥ	ケ	ノ	ル	♥	
A	LF	SB	*	:	J	Z	j	z			ェ	コ	ハ	レ	♦	
B	HM	EC	+	;	K	[	k	{			ォ	サ	ヒ	ロ	♣	
C	CL	→	,	<	L	¥	l	¦			ャ	シ	フ	ワ	●	
D	CR	←	−	=	M	]	m	}			ュ	ス	ヘ	ン	○	
E	SO	↑	.	>	N	∧	n	~			ョ	セ	ホ	゛		
F	SI	↓	/	?	O	_	o	DEL			ッ	ソ	マ	゜		DEL

注 意　SPは空白(スペース)コードを示す

■表 7-2　キャラクターコード表

ESC　[ps;…	;ps m	表示文字に属性を指示（既定の属性に戻すのはESC[m）
ps	(10 進)	
0		既定の属性
1		ハイライト（モノクロのみ）
5		ブリンク
7		リバース
16	8	シークレット
	30	黒　　淡（暗）
18	34	青
17	31	赤
19	35	紫
20	32	緑
22	36	水色
21	33	黄色
23	37	白　　濃（明）
	40	リバース黒
	41	赤
	42	緑
	43	黄色
	44	青
	45	紫
	46	水色
	47	白
		※モノクロ CRT 装置での色の指定は濃淡として処理

■表 7-3　CRT 制御コード

ASCII制御コード	1文字のASCII制御コードにより，CRT画面の制御を行う．
BEL　07	ブザーを約1秒鳴らす
BS　08	カーソルを1文字左に移動
HT　09	カーソルを次のタブ位置に移動 タブ位置　08，16，24，32，40，48，56，64，72
LF　0A	カーソルを同じカラム位置で1行下に移動
VT　0B	カーソルを同じカラム位置で1行上に移動
FF　0C	カーソルを1文字右に移動
CR　0D	カーソルを行の左端に移動
SUB　1A	CRT画面をすべてクリア（カーソルはホーム位置）
ESC　1B	エスケープコード
RS　1D	カーソルをホーム位置に移動
ESC　シーケンス	**CRT画面の制御を行うエスケープシーケンス．ここでESCはエスケープコード(1Bh)を表しパラメータpn，pl，pc，psは10進数を示す．**
ESC〔pl;pc H	カーソルをpl行，pc列に移動
ESC〔pl;pc f	ESC〔pl;pc Hと同様
ESC＝lc	ESC〔pl;pc Hと同様（パラメータlとcは20hのオフセットが加えられる）
ESC〔pn A	カーソルを同じカラム位置で上にpn行移動
ESC〔pn B	カーソルを同じカラム位置で下にpn行移動
ESC〔pn C	カーソルを右にpn行移動
ESC〔pn D	カーソルを左にpn行移動
ESC〔0 J	カーソルから，最終行の右端までをクリア
ESC〔1 J	先頭行の左端から，カーソル位置までをクリア
ESC〔2 J	CRT画面をすべてクリア
ESC　*	ESC〔2 Jと同様
ESC〔0 K	カーソルから，行の右端までをクリア
ESC〔1 K	行の左端から，カーソルまでをクリア
ESC〔2 K	カーソルの行の左端から右端までをクリア
ESC〔pn M	カーソルの位置する行から下にpn行削除し，以降の行を上に詰める
ESC〔pn L	カーソルの位置する行以降をpn行下に移動し，空白のn行を挿入
ESC D	カーソルと同じカラム位置で1行下に移動
ESC E	カーソルを1行下の行の左端に移動
ESC M	カーソルを同じカラム位置で1行上に移動
ESC〔s	カーソル位置とその表示文字の属性をセーブ
ESC〔u	ESC〔sでセーブした内容を戻す
ESC〔6 n	カーソル位置を直後のコンソール入力呼び出しにて知らせる．形式はESC〔pl;pc R
ESC)0	漢字を取り扱うモード
ESC)3	グラフ文字を取り扱うモード
ESC〔>5l	カーソルを画面上に表示するモード
ESC〔>5h	カーソルを画面上に表示しないモード
ESC〔>1h	CRT画面の最下位行をプログラムで使用可能とする
ESC〔>1l	CRT画面の最下位行はシステムで使用される
ESC〔>3h	CRT画面の表示行数を20行にする
ESC〔>3l	CRT画面の表示行数を25行にする

■表 7-4　MS-DOS 制御コード

8 プロの仕上げで これがプロとアマの違い

プロの作るソフトと私たちがつくるソフトと何が違うのでしょうか。実は、プロもアマチュアも内容的には変わらないプログラムを作っているのです。ただ、プロはお客用に外見の部分にかなりの力を注いでいます。他社のパッケージ・ソフトと競合してお客の心をつかむには、いったいどうしたらこのように作れるのかと不思議に思わせる魅力的な画面が重要な役割をはたします。

この章では私たちでもまねが十分できるやさしいプロのテクニックをいくつか紹介します。このテクニックとして、少ない労力で高い効果を生み出すものだけを選びました。なんだこんな程度かと思っていただければ結構です。これは一種のコロンブスの卵です。7章までのサンプル・プログラムもかなり利用できるはずです。

友人のプログラムと差をつけるのも楽しいものですし、TURBO Pascal はライセンス・フリー、つまり TURBO Pascal でコンパイルしたオブジェクト・プログラムは開発元の Borland International 社の許諾なしに自由に売ることができます。あなたが「こんなプログラムがあったら」とひごろ思っているプログラムは、きっと他のユーザーも欲しいプログラムです。ひとつここでがんばってそんなプログラムを開発し、ソフトハウスに売り込んでみるのもおもしろい体験になるでしょう。

なるほどこんな手がある

8-1■メニュー選択

パソコンのことがよくわかっていないユーザー向けにソフトを開発するときには、ユーザーのキーボード入力を簡単にするためメニュー方式をとるのが普通です。番号を付けたいくつかの項目を表示して、その中から希望のを選んで数字をキーボードから入力させるものが一番簡単です。しかし、簡単という意味は開発者にとってであって、ユーザーには面倒な方法です。それに見かけがあまりよくありません。

そこで利用したいのが TURBO Pascal の Ver.6.0 以降に強力なユーザーインタフェース開発のツールとして追加された TURBO Vision です。しかし、ちょっと簡単なメニュー選択を作るのに TURBO Vision を使うのはスズメを撃つのに大砲を使うようなものです。この節ではカーソル移動キーで大型のカーソルが移動するプログラムを紹介しま

す。このプログラムこそ、どう作ればよいのかまったく見当がつかないのに、やり方を一度知ってしまえばなーんだといいたくなるテクニックの代表です。ここでは、マルチプランなどのカーソルのように大きなリバースカーソルがメニュー選択時にカーソル移動キーで移動し選択するといったテクニックを紹介しましょう。簡単な割にはなかなか効果的なユーザーインタフェースが作れます。

まず、表示文字をリバースさせるルーチンを考えます。表示文字をリバースさせるには、CRT 画面の制御を行うエスケープ・シーケンスで、ESC［7；m　と表されます。TURBO Pascal で、エスケープ・シーケンスを使用するには、次のような方法でやります。エスケープコードは ASCII コードで 1BH で表されます。このコードを定数定義しておきます。こうしておけばあとはこれに続くパラメータを変えるだけで、エスケープ・シーケンスを使用することができます。つまり、

```
const
  ESC = #$1B;
     :
  write(ESC,'[7;m');
```

とすれば、これ以後の表示文字はリバースされて出力されます。このほかに本プログラムで使用するエスケープ・シーケンスコードは次のようなものです。

ESC[>5l	カーソルを画面上に表示するモード
ESC[>5h	カーソルを画面上に表示しないモード
ESC[7;m	表示文字をリバースに指示する
ESC[m	表示文字を既定の属性に戻す

この他のエスケープ・シーケンスについては、7-7 のランダムボックスを参考にしてください。エスケープシーケンスを使用しないでもまったく同じ効果を出せる手続きや関数が TURBO Pascal の Crt ユニットには用意されています。ここではまず、Dos ユニットによる例 (リスト 8-1) を示し、その後にエスケープシーケンスによる例 (リスト 8-2) を示します。

■リスト 8-1

```
 1: program LIST8_1;
 2:
 3: Uses
 4:     Crt;
 5:
 6: const
 7:     ESC          = #$1B;{エスケープ}
 8:     SPACE        = #$20;{スペース}
 9:     CR           = #$0d;{復帰}
10:     LF           = #$0a;{改行}
11:     MAX_SELECT   =    4;{選択肢－1}
```

```
12:
13: type
14:   SeLect_Type = record {表示用レコード}
15:                  XPos : integer;{表示カーソル位置}
16:                  YPos : integer;
17:                  item : string[60];{項目内容}
18:                end;
19:
20: var
21:   seLect : array[0..MAX_SELECT] of SeLect_Type;
22:   kekka : integer;
23:   Lst   : Text;
24: {-----------------------------------------------------------}
25: {リバース表示オン}
26: procedure On(seLect : SeLect_Type);
27: begin
28:   GoToXY(seLect.XPos,seLect.YPos);
29:   Write(Lst,ESC,'[7;m',seLect.item );
30: end;
31: {-----------------------------------------------------------}
32: {リバース表示オフ}
33: procedure Off(seLect : SeLect_Type);
34: begin
35:   GoToXY(seLect.XPos,seLect.YPos);
36:   Write(Lst,ESC,'[m',seLect.item);
37: end;
38: {-----------------------------------------------------------}
39: {選択肢作成・表示}
40: procedure Menu_BuiLder;
41: var
42:   i : integer;
43:
44: begin
45:   CLrScr;
46:   for i:=0 to MAX_SELECT do
47:   with seLect[i] do
48:   begin
49:     YPos := i + 10;{10行目の27文字目から}
50:     XPos := 27;
51:     item :=' どれを選びますか? No.'+Chr(i+49)+' ';
52:     Off(seLect[i]);
53:   end
54: end;
55: {-----------------------------------------------}
56: {メニュー選択}
57: function Get_SeLect : integer;
58: var
59:   ch     : char;
60:   CurPos : integer;
```

```
 61:
 62: begin
 63:   CurPos := 0;
 64:   On(seLect[0]);
 65:   repeat
 66:     ch := ReadKey;
 67:     if ch=SPACE then
 68:      begin
 69:        Off(seLect[CurPos]);
 70:        CurPos := CurPos + 1;
 71:      end;
 72:     CurPos := CurPos mod (MAX_SELECT+1);
 73:     On (seLect[CurPos]);
 74:   untiL ch = CR;
 75:   Get_SeLect := CurPos;
 76: end;
 77: {-----------------------------------------------------------}
 78: {メインプログラム}
 79: begin
 80:   Assign(Lst,'CON');{出力装置指定}
 81:   ReWrite(Lst);
 82:   Write(Lst,ESC,'[>5h');{カーソルを隠す}
 83:   Menu_BuiLder;
 84:   kekka := Get_SeLect+1;{ここでループする}
 85:   GoToXY(27,20);
 86:   Write(Lst,'あなたは',kekka:1,'番目を選びました');
 87:   Write(Lst,CR,LF); {復帰と改行}
 88:   Write(Lst,ESC,'[m');{表示文字の回復}
 89:   Write(Lst,ESC,'[>5L');{カーソルを表示する}
 90:   CLose(Lst);{出力装置使用終了}
 91: end.
```

それぞれの手続きの解説の前に、プログラムの中心となる56行目から始まる関数Get_Selectを見ておきましょう。ここではリバースカーソルがスペースキーによって5行からなるメニューを順番に移動することにしました。その条件に即してどのようにこの関数が動作するか考えてみます。リスト8-1を見てください。

CurPosは現在メニューの何行目にカーソルがあるかを示す変数です。ただし、一番上のメニューを0番目としています。プログラムが走りだした段階ではカーソルは一番上になければなりませんから、初期値を62行目で0にしておきます。63行目で最初にリバース表示する項目を0番目としておきます。リターンが入力されるまで、64〜73行目のrepeat 〜 untilでループします。65行目でReadKey関数によりキーボードから1文字をchとして手に入れます。ここではスペースキーをメニュー項目の移動用のキーにしていますが、ファンクションキー、DELキー、矢印キーなどのような特殊キーを使用するときには注意が必要です。PC-9801用のReadKey関数は漢字の入出力

が有効の場合、つまりCrtユニットで定義されているKanji変数が真のときとそうでないときではReadKeyが返すコードが違ってきます。Kanjiが真のときには特殊キーが発生するコードを直接返してくるので、キーの判定ができません。偽のときにはヌル文字(#0)と拡張スキャンコードを返してきます。このようなやっかいなことにはかかわらないことが一番ですからユーザーに操作させるキーはスペース、タブ、エスケープキーなどを使用するのがよいでしょう。

71行目はCurPosが0～4の間の数値を順々にとるようにするためのものです。72行目でカーソルが移動した項目をリバース表示します。73行目で押下キーが改行キーであればrepeatのループを抜け出します。74行目でカーソル位置をこの関数の値として返します。

以上のような関数Get_Selectの処理内容がわかってしまえば、あとは簡単です。リスト8-1を始めに戻って眺めてゆきましょう。6～9行目は定数宣言部です。ここでは主にASCII制御コードを定数として宣言しています。11行目から定義している型、Select_Typeはメニューの選択行を作るためのものです。XPosとYPosは画面の中でテキストを表示すべき座標です。itemはメニューとして表示すべき項目の内容です。

4つの手続き、On、Off、Menu_Builderは簡単ですから解説の必要はないでしょう。メインプログラムは78～85行の短いものです。

79行目でカーソルを画面から隠します。これに対応して84行目を必ず入れておきます。80行目でメニューを画面に表示し、81行目で何番目の項目を選んだかをえます。それを表示した後に、82行目で改行と復帰を行います。これがないとプログラムから抜けたときMS-DOSの表示がずれて見苦しくなります。

今度はMS-DOSのエスケープシーケンスを使って同じプログラムを作ります。リスト8-1Bを見てください。内部はほとんどリスト8-1と同じです。

■リスト8-1B

```
 1: program LIST8_1b;
 2:
 3: Uses
 4:   Crt;
 5:
 6: const
 7:   SPACE        = #$20;{スペース}
 8:   CR           = #$0d;{改行}
 9:   MAX_SELECT   =    4;{選択肢－1}
10:
11: type
12:   SeLect_Type = record {表示用レコード}
13:                   XPos : integer;{表示カーソル位置}
14:                   YPos : integer;
15:                   item : string[60];{項目内容}
16:                 end;
```

```
17:
18: var
19:   seLect : array[0..MAX_SELECT] of SeLect_Type;
20:   kekka : integer;
21: {-------------------------------------------------------}
22: {リバース表示オン}
23: procedure On(seLect : SeLect_Type);
24: begin
25:   TextReverse(1);
26:   GoToXY(seLect.XPos,seLect.YPos);
27:   Write(seLect.item )
28: end;
29: {-------------------------------------------------------}
30: {リバース表示オフ}
31: procedure Off(seLect : SeLect_Type);
32: begin
33:   TextReverse(0);
34:   GoToXY(seLect.XPos,seLect.YPos);
35:   Write(seLect.item)
36: end;
37: {-------------------------------------------------------}
38: {選択肢作成・表示}
39: procedure Menu_BuiLder;
40: var
41:   i : integer;
42:
43: begin
44:   CLrScr;
45:   for i:=0 to MAX_SELECT do
46:   with seLect[i] do
47:   begin
48:     YPos := i + 10;{１０行目の２７文字目から}
49:     XPos := 27;
50:     item :=' どれを選びますか？ No.'+Chr(i+49)+' ';
51:     Off(seLect[i])
52:   end
53: end;
54: {-------------------------------------------------------}
55: {メニュー選択}
56: function Get_SeLect : integer;
57: var
58:   ch     : char;
59:   CurPos : integer;
60:
61: begin
62:   CurPos := 0;
63:   On(seLect[0]);
64:   repeat
65:     ch := ReadKey;
```

```
66:     if ch=SPACE then
67:      begin
68:        Off(seLect[CurPos]);
69:        CurPos := CurPos + 1
70:      end;
71:     CurPos := CurPos mod (MAX_SELECT+1);
72:     On (seLect[CurPos]);
73:   untiL ch = CR;
74:   Get_SeLect := CurPos
75: end;
76: {------------------------------------------------------------}
77: {メインプログラム}
78: begin
79:   TextCursor(NoDispCursor);{カーソルを隠す}
80:   Menu_BuiLder;
81:   kekka := Get_SeLect+1;{ここでループする}
82:   GoToXY(27,20);
83:   Write('あなたは',kekka:1,'番目を選びました');
84:   TextCursor(DispCursor){カーソルを表示する}
85: end.
```

もうちょっと手を加えて

8-2■メニューボックス

8-1でメニューをリバース表示するプログラムを作りましたが、ちょっとこれだけではさびしい気がします。

そこで、メニュー用の簡単な枠を作ってみましょう。ここで作るメニュー用の枠、ボックスは7-7のランダム画面の作り方を参考にしました。こんな簡単な手続きでも使い方ひとつでメニュー画面をぐっと引き立たせることができるのです。

■リスト8-2

```
 1: Program  LIST8_2;
 2:
 3: Uses Crt;
 4:
 5: {座標 (x, y) に1文字書く}
 6: procedure PutChar(x,y,code:byte);
 7:   begin
 8:     GoToXY(x,y);
 9:     Write(Chr(code));
10:   end;
11:
12: {メニュー用の箱を描く}
```

```
13: procedure MenuBox(x1,y1,x2,y2:byte);
14: var
15:   a : byte;
16:   f : text;
17:
18:   begin
19:    kanji := FALSE; {グラフィック文字表示オン}
20:     CLrScr;
21:
22:     if (x1$<$2) or (x2$>$80) or (y1$<$2) or (y2$>$24) then
23:       Write('メニューボックスの指定位置が',
24:             '画面境界の外に出ています。')
25:    eLse
26:      begin
27:        for a:= x1 to x2 do
28:          begin
29:            PutChar(a,y1,$95);
30:            PutChar(a,y2,$95);
31:          end;
32:
33:        for a:= y1 to y2 do
34:          begin
35:            PutChar(x1,a,$96);
36:            PutChar(x2,a,$96);
37:          end;
38:
39:        PutChar(x1,y1,$98);
40:        PutChar(x1,y2,$9A);
41:        PutChar(x2,y1,$99);
42:        PutChar(x2,y2,$9B);
43:      end;
44:
45:    kanji := TRUE; {グラフィック文字表示オフ}
46:   end;
47:
48: {メインプログラム}
49:  begin
50:    MenuBox (10,3,70,20);
51:  end.
```

リスト8-2を見てください。6行目の手続きPutCharは文字を指定された画面上の場所(x,y)に表示するものです。メニュー用の枠を描く本体は13行目の手続きMenuBoxです。枠の左上と右下の対角になる角の座標を指定すると、グラフィック文字で枠を描きます。19行目でこれから表示する文字がグラフィック文字であることを指定します。これを忘れると、画面にとんでもない漢字が表示されます。

22行目でメニュー枠が画面内に納まるかどうかを判定し、はみ出るようであれば指

定を変更するようにメッセージを出します。ここでははみ出るときにはいきなりプログラムが終了してしまいます。画面内に納まる設定がえられるまで入力を要求するようにしてもよかったのですが、ここではあっさりと作ってあります。

27 ～ 43 行は7-7 のランダム画面と同じですから、文字のコードなどはそちらを参考にしてください。45 行目で再び画面表示を漢字に戻します。これを忘れると今度は漢字を表示しようとしたときに、とんでもない文字が飛び出します。メインプログラムは50 行目の 1 行だけです。(10,3) と (70,20) を対角線上の角にして枠を画面に描きます。

BASICにできてPascalに

8–3■PRINTUSING

TURBO Pascal には実数の数値を出力するときに、小数点以下の桁数を指定できる便利な機能があります。たとえば、

```
Write(1234.5678:6:3);
```

とすると、出力が□□ 1234.567 のようになります。しかし、ビジネス用に使いたい 3 桁ごとにカンマを打つ機能はありません。これがBASIC なら

```
PRINTUSING('###,###.###',1234.5678)
```

とすれば、□□ 1,234.567 のようになります。この程度で BASIC にいばられたのでは、TURBO Pascal も情けない存在です。そこで、TURBO Pascal で PRINTUSING と同じ機能を持つ手続きを開発してみましょう。リスト 8-3 を見てください。

■リスト 8-3

```
 1: program LIST8_3;
 2:
 3: Uses Crt;
 4:
 5: type
 6:   Str80 = string[80];
 7:
 8: var
 9:   number  : reaL;
10:   format  : Str80;
11:
12: procedure PrintUsing (format : Str80; number : reaL );
13: const
14:   COMMA = ',';
15:   POINT = '.';
16:   MINUS = '-';
17:
```

```
18: var
19:   format_Len,
20:   int_Len,
21:   degit_num,
22:   point_pos,
23:   i,j          : integer;
24:   comma_yes,
25:   point_yes,
26:   minus_yes    : booLean;
27:   print_out,
28:   integer_string : Str80;
29:
30: begin
31:   minus_yes := number $<$ 0;
32:   comma_yes := Pos (COMMA,format)$>$0;
33:   point_yes := Pos (POINT,format)$>$0;
34:   number    := abs(number);
35:   degit_num := 0;
36:   format_Len:= Length(format);
37:   if point_yes then {小数点}
38:     begin
39:       point_pos := Pos(POINT,format);
40:       degit_num := format_Len-point_pos;
41:     end;
42:   Str(number:0:degit_num,print_out);
43:   if comma_yes then {3桁コンマ}
44:     begin
45:      j:=0;
46:      integer_string:=Copy(print_out,1,
47:                  Length(print_out)-degit_num);
48:      int_Len:=Length(integer_string);
49:      if point_yes then
50:        int_Len:=int_Len-1;
51:      for i:=int_Len downto 2 do
52:        begin
53:         j:=j+1;
54:         if j mod 3 =0 then
55:         Insert(COMMA,print_out,i);
56: end;   end
57:     end;
58:
59:   if minus_yes then {負の数}
60:     print_out:=MINUS+print_out;
61:   Write(print_out:format_Len+1)
62: end;
63:
64: begin
65:   CLrScr;
66:   Write('形       式： ');
```

```
67:   ReadLn ( format );
68:   Write(' 数値データ： ');
69:   ReadLn(number);
70:   PrintUsing ( format, number );
71:   WriteLn
72: end.
```

12行目から始まる手続きPrintUsingがこのサンプルプログラムの中心です。引数のformatは##...#の表示形式、numberは表示したい実数です。手続きの本体は30行目から始まります。31、2、3行目で表示すべき形式を論理変数に入れておきます。たとえば、comma_yesという論理変数は、カンマイエスという変数名のとおり3桁ごとのカンマを指定したときです。

37～41行目で小数点が入っているときの処理を行います。42行目にある関数、Strの引数がちょっと見なれない形ですが、このような使い方も可能なのです。書式は、

```
Str(r:n:m,st);
```

で、実数rを幅nのフィールドに小数点以下m桁の形にして、文字列stに入れるということです。42行目ではnが0になっていますが、フィールド長が0のときには小数点を含むrの文字列としての長さが仮定されます。このことはマニュアルには書いてありません。実験的に見つけました。

43～56行のカンマの処理はもう少し複雑です。46行目では小数点を含む整数の部分をinteger_stringに入れます。51～55行で下から3桁ずつにカンマを挿入します。最後に59行目でマイナス記号を負の数に対しては付加します。60行目では文字列出力字の表示形式の指定を使っています。文字列に対する指定は、たとえば、

```
Write('ABCD':6);
```

とすれば、□□ABCDのように表示されます。ここでは表示に余裕を見込んで1文字だけ増やしてあります。もし、結果を印刷したいのであればWrite(Lst,...)のように、出力装置としてPRN、つまり印刷装置を指定します。

64～72行のメインプログラムの部分はもう説明しなくてもわかるでしょう。ここで開発した手続きPrintUsingをライブラリとして持っていれば、BASICとまったく同じ機能が実現できます。これをさらに改良するとすれば、¥1,234としたり***1,234.00のような出力ができるようにすることです。

いずれにせよ、プロのタッチのあるソフト開発には文字列の操作も大切です。

TURBO Pascalにはリスト8-3で使用したもの以外に、文字列用の標準関数と手続きが用意されています。それを活用できるようにもう一度マニュアルの文字列の部分を読んでみると、おもしろいプログラムを作るためのアイデアがきっとえられるでしょう。

9 メモリの壁を破る

Pascal を勉強していて、初心者が一番戸惑うのはポインタです。慣れてしまえばポインタはとても便利で強力な Pascal の概念なのですが、それが理解できないばかりにいつまでたっても BASIC の焼き直しのような Pascal プログラムを書いている羽目になっているひとが意外と多いのではないでしょうか。Pascal の初級と中級プログラマの違いは、いかに上手にポインタを利用するかです。

むろん、ファイルの取り扱いも結構むずかしいものですが、ここまでのトレーニングによってファイルの取り扱いは十分に理解できたと思います。そこで、この章では Pascal 理解のもうひとつの「壁」であるポインタ、つまり動的変数についてしっかりトレーニングしましょう。これさえマスターすればプログラミングに大きな幅ができ、Pascal の持つすばらしい機能をフルに発揮させることができるはずです。

ただ、ポインタがわかったというだけでは余り利用価値のあるプログラムは作れません。そこで、ポインタの応用問題として TURBO Pascal のメモリの壁ともいえるデータに対する 64K バイトの制限を破る大次元行列を扱うプログラムを開発してみます。

まずは復習から

9–1■静的変数 vs 動的変数

本題に入る前に、静的変数と動的変数について復習しておきましょう。動的変数については、この PUG ブックスシリーズの「TURBO Pascal トレーニングマニュアル」の第 11 章をあらかじめ読んでおくと、ずっと理解が早くなります。Pascal の変数には静的変数 (Static Variables) と動的変数 (Dynamic Variables) の 2 種類があります。

静的変数はプログラムで一度宣言してしまえば、そのために割り当てられたメモリのバイト数がプログラム実行中に変わることのない変数です。整数は 2、実数は 6、n 字のストリングの文字列は n ＋ 1、Byte 型なら 1 バイトなどとなります。このバイト長は TURBO Pascal のマニュアルに明記されています。

静的変数はコンパイル時にメモリ上に確保されます。もっとも、正確にはグローバル変数と型付き定数がコンパイル時に確保されます。関数・手続き内で宣言されるローカルな配列はその関数・手続きが実行されるときにのみスタック上に確保されます。した

がって、ローカルにしか使わない配列をグローバルに宣言するのは、メモリを消費するので感心しません。

さて、これに対して動的変数はコンパイル時に定義されますが、プログラムの実行時にしかメモリが割り当てられません。この動的変数がメモリ上のどこに割り当てられたのかを示すために設けられているのがPascalのポインタです。ポインタ自体はコンパイル時に確保され、これで動的変数の第1バイトを指すようにしています。

ここまで読んでくると、同じコンパイル時に確保されるのなら、ローカル変数で定義する静的変数と動的変数は違いがないのではないかという気がするでしょう。つまり、どちらもデータセグメント上のメモリを消費しないのだから、わかりにくい動的変数など使わないでも十分に大きな次元の変数を関数・手続き内で宣言すればメモリがパンクせずにプログラムが実行できそうです。

何故このようなわかりにくいポインタで指す動的変数が考えられたのでしょうか。その理由は、プログラム作成時にデータ数があらかじめ不明な場合の対策が必要だからです。たとえば、住所録を作るときを考えてみましょう。友人の数がある程度決っている人なら、何ページ位の住所録を買ってきてそれに記入すればよいでしょう。しかし、何人記入するか未定のときには、ページ数の少ない住所録ではパンクする恐れがありますし、多いものでは無駄を生じそうです。こんなときバインダ式の住所録があれば解決します。これと同じ考え方が動的変数なのです。

静的変数では最大のデータ量を予測して、安全を見越して配列などを大きめに取っておかねばなりません。たとえば、住所録のデータをレコード型にしたら、最大何人分用意するのかとなると、どうしても五百人、千人分の配列にしておかないと心配です。その大きな配列をプログラムのそこかしこで操作するとなると、グローバル変数にせざるをえません。そうすると、コンパイルの段階でメモリ不足になってしまいます。

それなら関数や手続きの中で使用のたびにローカル変数として定義しておけばよさそうですが、こんどは定義している関数や手続きが呼ばれたときにスタック・オーバーフローを起こして、プログラムが強制終了してしまいます。つまり、想定しうる最大のデータ量にメモリを確保するという方式がまずいのです。

動的変数はプログラム実行時にヒープに確保されます。16 ビットのパソコンでは、プログラムはセグメントと呼ばれる 64K バイトのメモリ領域を単位に構成されます TURBO Pascallではコードセグメント、データセグメント、スタックセグメントの3つを使用します。この3つを全メモリ量から引いた残りがヒープ領域になります。この領域をすべて使用すれば、かなり大きなデータが保持できそうです。

TURBO Pascal では、変数領域が 64K バイトに制限されている関係上、大きな配列を取ろうとしても 65,520 バイト、約 64K バイト以上取ることができません。たとえば、

```
program example;
var
```

```
a : array[0..100000] of integer;
```

というようにすると、コンパイル時にエラーが出てきてしまいます。

そこで、ポインタ型変数によりヒープ領域を使って64Kバイト以上の大次元配列をとる方法を紹介します。いきなり大次元配列の具体的なプログラムを提示してもわかりにくいと思いますので、最初にポインタの復習をかねたやさしい例題で説明しながら先へ進むことにしましょう。

初心に戻って

9–2■ポインタの復習

簡単なポインタ型変数を使った例をリスト9-1に示します。

■リスト9-1

```
 1: program LIST9_1;
 2:
 3: var
 4:     a:^integer;
 5:
 6: begin
 7:     New(a);
 8:     a^:=1234;
 9:     writeLn(a^);
10:     Dispose(a);
11: end.
```

4行目でaを整数型の動的変数として定義しています。そのときには約束としてinteger^ではなく^integerと宣言します。すぐ後ろの説明とも関連しますが、変数型の前につけられた^印は、その変数型のポインタ型であることを示します。そして、9行目で、動的変数a^に1234の整数を代入し、10行目でそれを呼び出して表示させています。

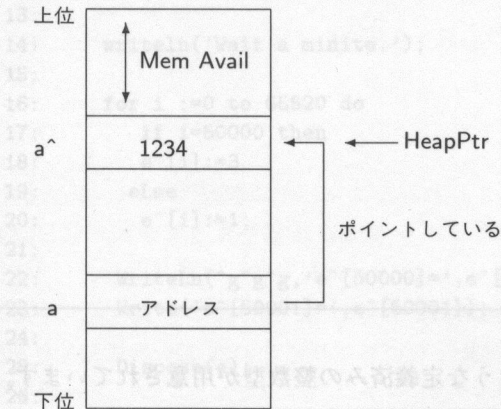
この中で^印(ハット、キャラット、サーカムフレックスなどと読みます)はポインタ型変数を意味します。aとa^では意味が次のように違います。

a　2バイト長の整数の第1バイトを指す(ポイントする)。
a^　aで指された整数の値。

したがって、9行目の意味はこうなります。

a^＝1234；　aの指すアドレスから始まる2バイトに整数値として1234を格納する。

手続きNew(a)により整数a^用に2バイトをヒープ領域に確保し、その第1バイトのアドレスをポインタ変数aに入れます。そして、Dispose(a)によってa^に使用され

ているヒープの使用を終了します。つまり、動的変数は使用のときだけメモリに確保されるのです。このプログラムでヒープにどのような変化が発生するのか図 9-1 に示します。図の中の HeapPtr は TURBO Pascal のシステム・レベル変数で、ヒープの最上位アドレスです。また、MemAvail はヒープ上の空きバイト数を返す TURBO Pascal の関数です。

■図 9-1　ポインタの概念図

これなら使えそう

9–3■1 次元配列

それでは配列型はどうすればよいのでしょうか。

ポインタ型変数を使って 1 次元配列を取った例がリスト 9-2 です。ここでは b を動的変数の配列として、4 行目と 7 行目で定義しています。まず、整数型の変数として 100 個の配列を定義し、それのポインタ型である配列として b を定義しています。これにより、b^[i] は整数型配列 b[i] により指される動的変数になります。

12 行目で、100 個の整数要素をヒープ上に確保し、13 行目で 1 から 100 までの整数を代入し、15 行目で再びそれを呼び出して表示させています。これを見ると、動的変数に値を代入したり、動的変数から値を持ってくるには、b^[i] という表現によって行われていることがわかるでしょう。

さらにこれを拡張し、64K バイトの 1 次元配列をヒープ上に確保してみましょう。正確には 64K バイトとは 65,536 バイトのことですが、TURBO Pascal では配列として取れる最大値は 65,520 バイトまでです。したがって、もしも整数型の配列を定義するのならこの 1/2 までになります。

■リスト 9-2

```
 1: program List9_2;
 2:
 3: type
 4:    ArrayType=array[0..99] of integer;
 5:
 6: var
 7:    b : ^ArrayType;
 8:    i : integer;
 9:
10: begin
11:    New(b);
12:    for i:=1 to 100 do
13:      b^[i]:=i;
14:    for i:=1 to 100 do
15:      write(b^[i],' ');
16:      Dispose(b);
17: end.
```

ところで、TURBO Pascal には次のような定義済みの整数型が用意されています。

型	範囲			形式
shortint	-128	–	127a	符号付き 8 ビット
integer	-32678	–	32767	符号付き 16 ビット
longint	-2147483648	–	-2147483647	符号付き 32 ビット
byte	0	–	255	符号なし 8 ビット
word	0	–	65535	符号なし 16 ビット

0 ~ 65535 の添字で配列を指すには、16 ビットで符号なしの整数表現が必要です。TURBO Pascal では、添字に一番よく使われる integer 型の整数は符号付きで－32768 ~ 32767 までしか表現できないので注意が必要です。整数の桁あふれはチェックされませんから、32767 から 1 越えると、今度は－32678 となってなかなか見つからないやっかいなバグのもとになりがちです。符号なしの整数表現はワード (word) 型で、0 ~ 65536 まで指すことができます。そこでここでは配列の添字として word 型を利用することにします。

その結果作ったプログラムがリスト 9-3 です。こののプログラムでは最大限のバイト数の配列を宣言しています。8 行目で配列 e の添字として使うべき i を word 型で宣言していることに注意してください。

■リスト 9-3

```
 1: program LIST9_3;
 2:
 3: type
 4:      ArrayType=array[0..65520] of byte;
 5:
```

```
 6: var
 7:    e     :^ArrayType;
 8:    i     : word;
 9:
10: begin
11:
12:    New(e);
13:
14:    writeln('Wait a minite.');
15:
16:    for i :=0 to 65520 do
17:        if i=50000 then
18:        e^[i]:=3
19:      eLse
20:        e^[i]:=1;
21:
22:      WriteLn(^g^g^g,'e^[50000]=',e^[50000]);
23:      Write('e^[50001]=',e^[50001]);
24:
25:      Dispose(e);
26:
27:   end.
```

プログラムはしごく簡単です。16 行目から i を 1 から 65520 まで増加し、もし i が 50000 のときには 3 を代入し、それ以外の時は 1 を代入しています。代入が終わったら i が 50000 のときと 50001 のときの数値を表示させます。このプログラムを実行するとブザーを 3 回鳴動させてから、3 と 1 が表示されます。このことから、e^[50000] と e^[50001] にアクセスできることが確認できます。

ぐんとステップアップ

9–4■2 次元配列

上の例で 1 次元配列、つまりベクトルは作れましたが、マトリックスである 2 次元行列を作るのはどうやったらよいでしょうか。TURBO Pascal の配列の制限内に入るはずの 255 行× 255 列の配列を定義する下のようなプログラムをコンパイルしてみてください。

```
program array\_test;
var
  a : array[0..254,0..254] of byte;
begin
end.
```

その結果得られるEXEファイルは1312バイトです。これはvarで配列を宣言しないとき、つまりbeginとendだけで何もしないプログラムのコンパイル結果と同じになります。TURBO Pascalは賢い(?)ので、使用されない変数である配列aのためのメモリ領域は確保されないのです。ところが、

```
program array\_test;
var
  a : array[0..254,0..254] of byte;
begin
  a[1,1] := 0;
end.
```

のようにプログラム内部で何かをさせようとするとコンパイル段階でエラーが出ます。配列aはグローバル変数ですからデータセグメント上に確保されますが、そこには収容しきれないからです。

つまり、普通に宣言したのでは2次元配列はせいぜいinteger型でも100行×100列程度しかつくれません。そこでポインタ変数を利用してメモリの許すかぎり大きな配列を確保するテクニックを紹介します。

■リスト 9-4

```
 1: program LIST9_4;
 2:
 3: type
 4:     ArrayType=array[0..1000] of byte;
 5:     ArrayPointer =^ArrayType;
 6:
 7: var
 8:     a  :array[0..1000] of ArrayPointer;
 9:     i,j:integer;
10:
11: begin
12:     WriteLn;
13:     WriteLn('Available Memory (before) = ',Memavail);
14:     WriteLn;
15:     WriteLn('///// ALLocating heap space./////');
16:     for i:=0 to 400 do
17:       New(a[i]);
18:     WriteLn;
19:     WriteLn('Available Memory (after)= ',Memavail);
20:     WriteLn;
21:     WriteLn('///// Making unitary Matrix./////  ');
22:     for i:=0 to 400 do
23:      for j:=0 to 400 do
24:        begin
25:          if i=j then a[i]^[j]:=1
26:            eLse
```

```
27:           a[i]^[j]:=0;
28:        end;
29:
30:    WriteLn('///// ReLeasing heap space. /////');
31:
32:    for i :=0 to 400 do
33:        Dispose(a[i]);
34:    WriteLn;
35:    WriteLn('Available Memory (final)= ',Memavail);
36:
37: end.
```

リスト 9-4 を見てください。

ここでは例として 400 行× 400 列＝ 16 万要素の 2 次元バイト配列を作っています。3 行目の type 宣言部で、まず ArrayType として 1000 要素の 1 次元バイト配列を用意しておきます。この配列をさらに 1000 個作れるようにしておけば、「理論的」には 1000 行× 1000 列の 2 次元バイト配列がヒープ上に作れるはずです。そのための準備が 5 行目で行っている ArrayPointer を ArrayType の配列としているところです。

この宣言は 8 行目の配列 a を ArrayPointer の配列として宣言しているところと連動しています。これで 1000 個の要素のある配列を 1000 個定義したことになります。それを 9 行目の integer 型の i と j で指すようにします。ここでは添字は最大で 1000 までですので整数型でかまいません。

具体的にどうしてこれで 2 次元配列になるのかはリストの先を見て行くのが早いようです。13 行目でヒープ上で使用可能なメモリ量を表示します。16 行目でまず a[i] を 400 個ヒープ上に New により確保します。ただし、a[i] は 8 行目で定義した変数ですから、400 バイトが確保されるのではなく、a[i]1 個につき 400 バイトが確保され、16 行目で 400 × 400 = 16 万バイトがヒープ上に確保されるのです。

この操作が終わった段階であとどれだけヒープ上に使用可能メモリが残っているのかを 19 行目で表示します。実際の要素へのアクセスは 22 ～ 28 行目を見てください。ここでは対角要素が 1、そのほかの要素が 0 である単位行列を作っています。単位行列とは、

1 行	100 ⋯ 00
2 行	010 ⋯ 00
3 行	001 ⋯ 00
	⋯
399 行	000 ⋯ 10
400 行	000 ⋯ 01

のような配列です。

この配列の i 行 j 列目は a[i]^[j] としてアクセスできます。25 行目では i と j が等しいときにはそれを 1 とおき、そうでないときには 27 行目で 0 としています。32 行目で

a[i] に確保したヒープを解放します。これをやらずにプログラムを終了しても TURBO Pascal はシステム、つまり MS-DOS にメモリを返します。したがって、ここでは確かに Memavail が元の値になったことを確認する意味しかありません。しかし、実際にはこの例題のような単純なケースはまず存在しません。配列に対して各種演算を加え、その配列がいらなくなったら Dispose でヒープから捨てるというのが一般的です。

このリスト 9-4 を TURBO Pascal の統合環境で実行させると 16 行目の実行中にメモリがパンクしてエラーが出て停止してしまいます。C(コンパイル) オプションでディスク上に EXE ファイルを生成するように指定してコンパイルし、TURBO Pascal から MS-DOS に抜けてから実行してください。

ここで紹介したポインタの使い方はちょっとしたノウハウです。TURBO Pascal に付属するユーザーズガイドにはポインタの基本的な使い方だけがあっさりと書いてあります。また、このガイドには「すぐれた Pascal の参考書」を読むようにと書いてあります。本書がすぐれているというのはいい過ぎですが、この章でポインタの使い方については理解できたのではないでしょうか。

プログラム全体を通して大次元行列が必要なことはあまりありません。大次元行列に対して処理を加える直前にヒープ上に設定し、処理が終ったら元にヒープを戻しておけば、そんな行列を実装メモリの許す限りいくつでも作れます。

ここでは整数の大次元行列をどのように作るかを考えてきましたが、科学計算では実数配列をよく使います。実数となると 1 個で 6 バイト使いますから、整数の 3 倍になります。こんなアプリケーションにこそこの章でトレーニングしたポインタを活用したいものです。TURBO Pascal ではデータは 64K バイトに制限されていますといわれると、すぐに大きな行列を使う計算はだめだと決めつけず、動的変数で解決するようにすべきです。

必要は発明の母ともいうではありませんか。あれがない、これがないとグチをいわず、どうやったら自分の希望する機能を TURBO Pascal で実現できるかをよく考えるべきです。結局、それがプログラム上達の道なのですから。

10 通信のしくみを Pascal で

最近、パソコン通信が盛んに行われるようになり、パソコンのワープロと並ぶもうひとつの大きな分野はネットワーク利用といわれています。大きな商用のデータベースをアクセスするにもパソコンはソフトウェア次第で、強力な通信端末としても利用できます。それでは、TURBO Pascal では通信をサポートするためどんな機能が用意されているのでしょうか。

結論から先にいうと、通信に対して特殊なものは何も用意されていません。その理由は、通信自体をまったく特殊なものとは扱わず、一般的なファイルに対する入出力とみなしているからです。イメージ的に通信回線の向こうにあるパソコンは一種の外部装置と考えているからなのです。

たしかに、通信特有の BIOS や割り込みなどがありますが、通信の主要部分である送受信は、装置指定を Aux として Read(Ln) あるいは Write(Ln) すればよいのです。つまり、プリンタに出力するときに PRN と指定したのと同じ感覚で扱うことができます。

この章ではそれだけでは済ますことができないプログラムの開発に必要な RS-232C を利用するためのユニットを開発します。ここで紹介するプログラムは TURBO Pascal のディスクにアセンブラで MSA 社から提供されているサンプルプログラムを、RS-232C 関係に使いやすくするために TURBO Pascal に移植した簡単なものです。

しかし、簡単だからといっても実用性がないわけではありません。これを利用すれば通信ソフトの開発や、RS-232C を入出力ポートとして外部機器を接続するようなことも楽になります。したがって、ここでは通信用の入出力ポートである RS-232C を制御するためのツール (道具) を作ります。

まずはインタフェースを理解して

10–1■RS-232C

パソコンの通信に絶対的に必要なのは RS-232C のインタフェースです。PC-9801 は本体背面にプリンタ用のソケットと並んで RS-232C ソケットが設置されています。これがないパソコンだとオプションを購入して増設スロットを利用して RS-232C ソケットを設けます。RS-232C はアメリカの EIA(Electronic Industries Association) が定め

た直列データ伝送用のインタフェース規格です。パソコンに使われるようになってから、この規格がパソコン通信の世界的な規格になってしまいました。事実、ほとんどのパソコンにはRS-232Cが標準で用意されています。

通信のためには、RS-232Cのハードウェアについての知識は「ある程度」必要です。自動車の運転を例にとってみると、日本では免許を取ってからまず一度も触れもしない自動車の構造について勉強が義務づけられていますが、アメリカではそんなことは一切やりません。そのかわり、いかに法規を守って安全運転してカーライフをエンジョイするかに重点が置かれています。つまり、ユーザーはユーザーに徹するという考え方なわけです。パソコン通信でもできあいのソフト・パッケージを購入して使う立場に徹してしまえば、通信がどのようにおこなわれるのか知る必要はありません。しかし、通信ソフトウェアをTURBO Pascalで開発するとなると、パソコン通信について最低限の知識が必要になります。そこで、ここではごく簡単なパソコン通信ソフトを作るための知識について簡単に説明しておきます。

パソコン通信のソフト開発にどうしても必要なものは、パソコン2台とその両方を接続するケーブルです。このケーブルはいわゆるRS-232C用ケーブルではありません。クロス・ケーブルと呼ばれるケーブルです。普通のRS-232Cケーブルでは一方のパソコンの送信端子ともう一方の送信端子が接続されてしまいます。同じことが受信端子についてもおこります。つまり、パソコン同士がお互いの送信端子に対して情報を送りだし、受信端子から受信しようとするとおかしなことになってしまいます。ちょうど電話機の受話器を口に当て、送話機を耳に当てる状態になってしまうのです。

これを解決するには2台のパソコンの受信端子と送信端子をクロス(交差)させて接続しなければなりません。クロスさえできれば特殊なケーブルは必要ありません。そのための小道具としてリバースアダプタというものがあります。RS-232Cのソケットに差しこめるようになっていて、後ろがRS-232Cのソケットと同じものがついている装置です。こう言われてもぴんとこないでしょうから、パソコンショップなどで実物を見せてもらうとよいでしょう。このリバースアダプタがあると普通のRS-232C用ケーブルをクロスケーブルとして使用できます。

ここまで読んできて、パソコン通信にはモデムが必要なはずだと思われたのではないでしょうか。確かに、実際にパソコン通信は電話回線を使用しますから、モデムがなければどうしようもありません。しかし、モデムによる電話回線を利用するパソコン通信では、電話機が2台必要です。おまけに、開発中は通信をテストするたびに電話を一方から他方へかけ、モデムの動作モードを設定したりする操作が必要となり面倒なことになりますから、開発のペースががっくりと低下してしまいます。パソコンを2台そろえるだけでもアマチュアには大変なことなのに、その上に電話機を2台(2回線)とモデムも2台というのは現実的ではありません。ぜひクロスケーブルかRS-232Cケーブルとリバースアダプタを用意してください。

その他に知っておきたいのはパリティチェックとフロー制御です。

まず、パリティチェックですが、これは偶数(イーブン)と奇数(オッド)の2種類あることと、通信中の雑音により誤った情報が伝送されるのを防止するための約束ごとだという程度だけ知っておけばよいでしょう。フロー制御はパソコン同士で相手が受信できない状態のときに「たれ流し」的に次々に送信がおこらないようにするためのパソコン通信上の規則です。

一方のパソコンが送信しているときに、受信側のパソコンの処理速度が遅いと送信した情報が捨てられてしまいます。そんなことにならないように、受信側がもう受信用に用意したバッファの3/4を越える文字が受信されると、送信側にたいしてX-OFF(Ctrl-S = 13H)信号を送り、送信側のパソコンに送信を一時中断するように指令します。受信側がバッファから文字を取り出し、バッファの1/4以下しか文字がなくなったらなったらX-ON(Ctrl-Q = 11H)信号を送り、送信を再開するように指令します。

もっともフロー制御したからといって、上のような動作が自動的に行われるわけではありません。送信側がX-ONやX-OFF信号を認識して送信を止めたり再開してくれたりするようにプログラミングされていなければならないのです。このフロー制御は通信速度の遅い場合には必要ありませんが、高速で通信するときには必要になります。通信速度がどの程度だと遅く、どの程度だと高速かは今のところ大体の目安として300BPSなら超低速、1200BPSで標準、2400BPS以上なら高速と思ってよいでしょう。

ここで、BPSとは通信速度の単位でBit Per Second(ビット・パー・セカンド)という1秒当り伝送される通信ビット数です。1万円前後の安価なモデムでは1200BPSが普通で、パソコン通信でもこの伝送速度が一番多く用いられています。300BPSだと上に説明したフロー制御はまず必要ありません。

2400BPSだと一応満足できる伝送速度が実現できますが、受信側の処理速度を大幅にスピードアップさせなければなりません。TURBO Pascalでも処理速度が問題になりますから、頻繁に使う手続き類はアセンブラで開発するなどかなりやっかいなことになります。もちろん、フロー制御すれば受信側の処理速度が遅くても2400BPSで通信することは可能です。

BIOSを調べてみる

10–2■RS-232CのBIOS

高級言語でプログラムを書く理由のひとつに異機種間のプログラム移植性を高めることがあります。しかし、現在のパソコンの処理速度では高級言語では十分な処理速度が得られない場合があります。それを解決するためにアセンブラでプログラム開発するということになります。現在の16ビットパソコンの主流である8086系のCPUであれ

ば、アセンブラは共通していますしMS-DOSのアセンブラMASMで開発すれば移植性もかなりあります。

ところが、アセンブラは共通でもそれで作ったプログラムはハードウェアにかなり依存しますから、アセンブラとしては移植できてもプログラム内容はパソコンによって大きく異なってしまいます。おまけに、アセンブラでプログラム開発するためにはアセンブラの勉強も必要です。そこで便利に使われるのがMS-DOSのシステム・コールです。高級言語とシステム・コールを使えば異機種間の移植性はほとんど100%までになります。しかし、MS-DOSの通信関係のシステム・コールは03、04番の補助装置に対する入出力のみというさびしさです。

しかし、パソコンに用意されているBIOSを利用すれば移植性は低下しますが、アセンブラでプログラム開発をする手間が省けます。もちろん、BIOSはパソコンの機種に依存するものですが、移植にあたってはこの部分だけを書き直すだけで済みますから、アセンブラよりはるかに移植が楽になります。そこで、移植性を少々犠牲にしても高速な通信プログラムをTURBO Pascalで開発したい読者のためにPC-9801に用意されているRS-232C関係のBIOSを紹介しておきます。BIOSはIntr手続きによって次のように呼び出して使用します。

```
Intr($19, Regs);
```

ここで、RegsはDosユニットで定義されたRegisters型の変数であるとします。

(a) RS-232Cの初期化

【入力条件】

AH ~ 00H

AL ~伝送速度 (表10-1)

BH ~ 500m秒単位の送信タイムアウト時間指定で、01H(500m秒) ~ FFH(127.5秒)、00Hなら1秒

BL ~ 500m秒単位の受信タイムアウト時間指定で、01H(500m秒) ~ FFH(127.5秒)、00Hなら30秒

CH ~ μPD8251A(RS-232Cインタフェース) モード設定情報 (表10-2)

CL ~ μPD8251Aコマンド指定 (表10-3)

DX ~受信バッファサイズ (表10-4)

ES : DI ~受信バッファ先頭アドレス (表10-4)

【出力条件】

AH ~リターンコード (正常終了時は00H)

(その他入力条件のレジスタ以外は保存)

(b) フロー制御を伴う初期化

【入力条件】

AH ~ 01H
その他は (a) と同じ。
【出力条件】
(a) と同じ。
ただし、表 10-4 の FLAG の XON ビットが 1 にセット。

(c) 受信データ長入手 (GET DTL)

【入力条件】
AH ~ 02H
【出力条件】
AH ~リターンコード
　00H：正常終了
　01H：RS-232C の初期化がされていない
　02H：受信バッファオーバーフロー
CX ~データ・ステータスの 2 バイトを 1 ワードとした受信ワード数
(その他のレジスタは保存)

(d) データ送信 (SEND　DATA)

【入力条件】
AH ~ 03H
AL ~ 1 バイトの送信データ
【出力条件】
AH ~リターンコード
　01 ~ 02H：(c) と同じ
　03H：μ PD8251 が送信可能状態 (TXRDY) でない

(e) データ受信 (RECEIVE DATA)

【入力条件】
AH ~ 04H
【出力条件】
AH ~リターンコード ((d) と同じ)
CH ~受信データ。
バッファが空のときはタイムアウトまで待ち、それでも空ならばリターンコードに 03H(受信可能 RXRDY ではない) を出力する。
CL ~受信データの受信時のステータス (フロー制御有効時に X-OFF 出力していて、CNT がバッファサイズの 1/4 以下になると、自動的に X-ON を相手側に出力する)。

(f) コマンド出力 (COMMAND OUT)

【入力条件】

AH ~ 05H

AL ~ μ PD8251 へ出力するコマンド (表 10-3)

【出力条件】

AH ~リターンコード ((d) と同じ)

(AX 以外のレジスタは保存される。IR 指定だと FLAG の INIT ビットがセットされ、初期化待ちの状態になる。)

(g) RS-232C ステータス入手

【入力条件】

AH ~ 06H

【出力条件】

AH ~リターンコード ((d) と同じ)

CX ~表 10-5

■ (旧版の表 10.1 ~ 5) ※巻末参照

それではユニット作成

10–3■RS-232C 用ユニット

ここまでの知識で TURBO Pascal で提供されているオリジナルサンプルプログラムの RS232C.PAS、RS232C.ASM をもっと使いやすく TURBO Pascal に移植します。この 2 本のプログラム・ファイルはディレクトリ ¥TP¥UNITS の中に入っています。このオリジナルをまず最初にコンパイルし、次に同じディレクトリの中に入っている RSDEMO.PAS をコンパイルするという順序になります。実際に動かしてみるときには、RSDEMO.PAS の頭部に書いてあるように、MS-DOS のバージョンが 3.10 以上であることと、CONFIG.SYS の中にデバイスドライバ RSDRV.SYS が入っていないことを確認してからにします。

さらに、プログラム頭部で「サンプルプログラムなので改良や技術的なサポートはしない」むね書いてありますから、これを改造したり参考にしたりするにはどうしても RS232C.ASM というアセンブラで書かれたプログラムを解読しなければなりません。しかし、後々の利用やプログラムの読みやすさ、改造の容易さのためには TURBO Pascal でこの部分を書いておいたほうが楽です。そこで MSA 社のオリジナルを移植してみたのがリスト 10-1 です。

このリストではオリジナルの 2 本のファイルを TURBO Pascal の RS232C ユニットとして 1 本にまとめてあります。ただし、コメントはできるだけオリジナルのものに合わせてあるので、リスト 10-1 と比較してみてください。

■リスト 10-1

```
 1: unit RS232C;
 2:
 3: {$D-,I-,S-}
 4:
 5: interface
 6:
 7: const
 8:   TRANS_75   = $00;             { 通信速度 75BPS }
 9:   TRANS_150  = $01;             { 通信速度 150BPS }
10:   TRANS_300  = $02;             { 通信速度 300BPS }
11:   TRANS_600  = $03;             { 通信速度 600BPS }
12:   TRANS_1200 = $04;             { 通信速度 1200BPS }
13:   TRANS_2400 = $05;             { 通信速度 2400BPS }
14:   TRANS_4800 = $06;             { 通信速度 4700BPS }
15:   TRANS_9600 = $07;             { 通信速度 9600BPS }
16:   XFLW_ON    = $01;             { Xフロー制御をする }
17:   XFLW_OFF   = $00;             { Xフロー制御をしない }
18:
19: var
20:   RsResult       : Byte;        { 手続きの実行結果 }
21:   RsRecvStatus   : Byte;        { 受信データの受信時ステータス }
22:   Rs8251AMode    : Byte;        { μPD8251モード設定値 }
23:   Rs8251ACommand : Byte;        { μPD8251コマンド指定値 }
24:   RsSendTimeOut  : Byte;        { 送信時タイムアウト時間 }
25:   RsRecvTimeOut  : Byte;        { 受信時タイムアウト時間 }
26:
27: procedure RsInit(trans: byte; buf: pointer; buf_siz: word;
28:                  x_flg: byte); { 初期化 }
29: procedure RsTerm;               { 終了処理 }
30: procedure RsPutc(chr: Char);    { 文字の送信 }
31: function RsGetc: Char;          { 受信文字取得 }
32: function RsLoc: word;           { 受信文字数取得 }
33:
34: implementation
35:
36: uses DOS;
37:
38: const
39: RS_BIOS      = $19;             { BIOS ファンクション番号     }
40: INIT1        = $00;             { フロー制御を行わない初期化 }
41: INIT2        = $01;             { フロー制御を行う初期化     }
42: LOC_RECVDATA = $02;             { 受信データ長の取得         }
43: SEND_DATA    = $03;             { データの送信               }
44: RSCV_DATA    = $04;             { データの受信               }
45:
46: var
47:   TRANS_speed,                  {通信速度}
48:   XFLOW_FLG   : BYTE;           {Xフロー制御}
49:
```

```
50: {-----------------------------------------------------------}
51: procedure RsInit(trans: byte; buf: pointer;
52:                      buf_siz: word; x_flg: byte);
53: type
54:   DWord = record
55:     HighWord ,
56:     LowWord   : word;
57:   end;
58:
59:
60: var
61:   Regs : registers;
62:   ptr  : DWord absolute buf;
63:
64: begin
65:   TRANS_speed := trans;
66:   Regs.AL := trans;
67:
68:   Regs.ES  := ptr.HighWord;    { buf address }
69:   Regs.DI := ptr.LowWord;
70:   Regs.DX := buf_siz;
71:   XFLOW_FLG := x_flg;
72:
73:   if x_flg = XFLW_ON then
74:      Regs.AH := INIT2
75:    else Regs.AH := INIT1;
76:
77:   Regs.BH := RsSendTimeOut;
78:   Regs.BL := RsRecvTimeOut;
79:   Regs.CH := Rs8251AMode;
80:   Regs.CL := Rs8251ACommand;
81:
82:   intr( RS_BIOS, regs );
83:   RsResult := Regs.AH       {リターンコード}
84: end; { of RsInit }
85:
86: {-----------------------------------------------------------}
87: procedure RsTerm;
88: var
89:   Regs : registers;
90:
91: begin
92:   Regs.DI := 0;          { buf offset address clear }
93:   Regs.ES := 0;          { buf segment address clear }
94:   Regs.DX := 0;            { buf_siz clear }
95:   Regs.AL := TRANS_speed;    { trans }
96:
97:   if XFLOW_FLG = XFLW_ON then
98:      Regs.AH := INIT2
```

```
 99: else Regs.AH := INIT1;
100:
101:   Regs.BH := RsSendTimeOut;
102:   Regs.BL := RsRecvTimeOut;
103:   Regs.CH := Rs8251AMode;
104:   Regs.CL := Rs8251ACommand;
105:
106:   intr( RS_BIOS, Regs );
107:   RsResult := Regs.AH;      { リターンコード }
108: end; { of RsTerm }
109:
110: {--------------------------------------------------------------}
111: procedure RsPutc(chr: Char);
112: var
113:   Regs : registers;
114:
115: begin
116:   Regs.AL := byte(chr);      { 送信文字 }
117:   Regs.AH := SEND_DATA;
118:     intr( RS_BIOS, Regs );
119:   RsResult := Regs.AH;      { リターンコード }
120: end; { of RsPutc }
121:
122: {--------------------------------------------------------------}
123: function RsGetc: char;
124: var
125:   Regs  : registers;
126:
127: begin
128:   Regs.AH := RSCV_DATA;
129:   intr( RS_BIOS, Regs );
130:
131:   RsResult := Regs.AH;      { リターンコード }
132:   RsRecvStatus := Regs.CL;  { 受信データの受信時ステータス }
133:   RsGetc := char( Regs.CH );  { 受信データ }
134: end; { of RsGetc }
135:
136: {--------------------------------------------------------------}
137: function RsLoc: word;
138: var
139:   Regs  : registers;
140:
141: begin
142:   Regs.AH := LOC_RECVDATA;
143:   intr( RS_BIOS, Regs );
144:
145:   RsResult := Regs.AH;      { リターンコード }
146:   RsLoc := Regs.CX;      { 受信データ長 }
147: end; { of RsLoc }
```

```
148: {
149: {=============================================================}
150:
151: begin
152:   { 各変数の初期値を設定する }
153:   Rs8251AMode    := $4E;
154:                  {ストップビット:1, データビット:8,パリティ:イネーブル}
155:   Rs8251ACommand := $37; {コマンド指定}
156:   RsSendTimeOut  := $02; {デフォルト時間: 1秒}
157:   RsRecvTimeOut  := $1E; {デフォルト時間: 15秒}
158: end.
```

それではリスト10-1の中を見てみましょう。まず、1行目でユニット名の宣言をします。このユニット名はデモ用プログラムRSDEMO.PASで使用することになっているユニット名RS232Cに合わせてあります。3行目はこのユニットの実行速度を向上させるためのコンパイラ指令です。

このユニットの外部とのインタフェースが5～32行目です。8～15行目までの定数は通信速度を指定するためのものです。16、17行目でXフローオンとオフの場合のそれぞれ定数の定義しておきます。19～25行目で宣言されている変数の意味はリスト内のコメントを参考にしてください。

27～32行目はこのユニットが提供する手続きと関数の宣言です。ここには3つの手続きと2つの関数があります。RS-232Cの初期化をするRsInit、使用の終了に伴う処理をするRsTerm、文字送信をするRsPutcと受信をするRsGetc、そして受信した文字数を手に入れるためのRsLocです。

これら手続きと関数の具体的内容が34行目以下の実現部です。このUnitではBIOSコールを使いますから、DOSユニットの使用を36行目で宣言しておきます。38～48行目で宣言されている定数、変数の意味はリスト中のコメントを参照してください。

51行目から始まるRsInitはRS-232Cの初期化です。引数としては通信速度のtrans、バッファの先頭を示すポインタbuf、バッファのサイズbuf_siz そしてXフロー制御のオンオフフラグx_flgがあります。53行目で定義している型DWordは2バイトをひとつとして取り扱い、しかもその上位バイトと下位バイトを別々にアクセスしたいときによく使うテクニックです。

61行目の変数RegsはもうおなじみのDosユニットのレジスタを示す変数です。62行目では変数ptrをバッファbufの先頭アドレスに一致させるようにしています。65行目からは83行目でRS-232CBIOSコール$19を行うための準備です。82行目でRS_BIOSとなっていますが、リストの先頭のほうでこれは$19である定数としています。

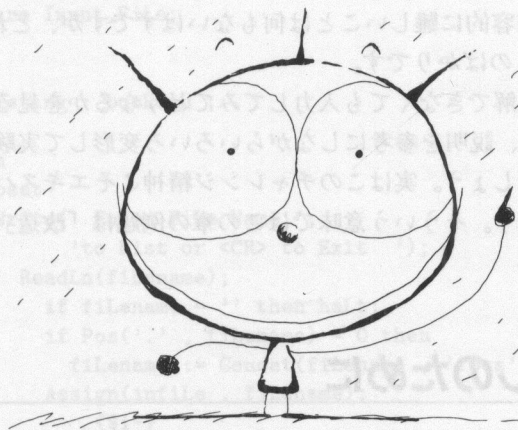
このように定数をわざわざ文字で現しておくのは、こうすればこのユニットを他のパソコンに移植しいたときなど、プログラムの先頭のほうに置いた定数宣言部の値を変えるだけで済むからです。さもないと、このユニットの中にある$19をすべて探さな

ければならなくなります。また、もしも$19がある特定の文字を表していたりすると、ユニット中に現れる$19がどちらを意味するのかいちいち考えなければならなくなります。こんな無用なことを避けるため、定数はできるだけそのものの値でなく、わかりやすい名前を付けて置くことをお勧めします。

87行目からの手続きRsTermはRS-232Cの使用を終了するための処理ですが、内容自体は初期化のための処理とほとんど変わりません。ただバッファに関係するものをすべて0にリセットしているところが違ってきます。11行目からの手続きRsPutcは1バイト、つまり1文字を送信します。123行目の関数RsGetcはバッファから1文字を取り出します。137行目の関数RsLocは受信データ長を手に入れるための関数です。

以上の手続きや関数は前節のRS-232CのBIOSのまとめを参考にすれば何をしているのか一目瞭然でしょう。151行目からμPD8251Aの初期化とデフォルトの送信タイムアウト時間、受信タイムアウト時間を設定します。

これでRS-232C用のユニットRS232Cができました。このユニットをコンパイルしてRS232C.TPUを作ってから、TURBO Pascalの中に入っているRSDEMO.PASをコンパイルしRSDEMO.EXEをディスク上に作ってください。正しくプログラムが動作するかどうかを調べるにはパソコンを2台RS-232Cクロスケーブルで接続してみるのが一番です。クロスケーブルを持っていなければパソコンとモデムを接続しているストレートケーブルにリバースアダプタをつないでクロスケーブルにするのがよいでしょう。リバースアダプタは千円程度ですから、ひとつ手にいれておくことをお勧めします。RS-232C関係のプログラムを作るときにとても重宝します。

ここではPC-9801でこのプログラムを走らせ、クロスケーブルでDynaBookに接続しました。DynaBook側は市販の通信ソフトで通信速度を9600BPSに設定してみたところ、あたりまえのことですが見事に動作が確認できました。

11 快適プログラミングのためのノウハウ

さらなる発展のために

第 10 章までにエキスパートになるための一通りのテクニックをトレーニングしました。しかし、ただ例題として紹介したプログラムを写して、それを自分のパソコンで TURBO Pascal により走らせてみても、それはそれだけのことです。そんなことなら初心者でも正確にタイプ入力できればよいだけの話です。それだけでは TURBO Pascal のエキスパートになるのはおぼつきません。

この本で提供されている例題は初めにお断りしたように、あくまでも出発点にすぎません。したがって、この章まで読み進んでこられた読者のあなたほどの「実力者」ならば、例題のいくつかは例のための例になっていて実用性がないじゃないかという不満があるでしょう。けれども、なるほどこんなことが TURBO Pascal でプログラミングできるのかということがわかれば、そこを出発点にしてすばらしいプログラムへと発展できる可能性があります。実はそこへ発展できるような例題をできるだけ取り上げてきたつもりです。

ここまでのトレーニングによりいろいろなプログラム開発を通して、いかに TURBO Pascal の持っている機能をフルに発揮させるかを試みてきました。実はそのことを味わうことがエキスパートへの道です。この章ではここまでのトレーニングの分類に入らなかったもので、きっと役に立つと思われるプログラムの開発とテクニックの紹介をしてみたいと思います。内容的に難しいことは何もないはずですが、どれもちょっと知っているととても便利なものばかりです。

プログラムの内容が理解できなくても入力してみてどうなるかを見るだけでもおもしろいと思います。むろん、説明を参考にしながらいろいろ変形して実験してみても新しいアイデアがえられるでしょう。実はこのチャレンジ精神こそエキスパートになるために最も大切なことなのです。そういう意味ではこの章の例題は「改造」のしがいのあるものばかりです。

リスト打ち出しのために

11–1■リスター

あなたはプログラムをデバッグするときにパソコンの画面でやりますか、それとも

いったん印刷して紙の上でやりますか。むろん、パソコンの画面上でデバッグできればそれにこしたことはありませんが、せいぜい20数行しか見通せないので、どうしても紙に印刷せざるをえないのが実状です。

紙を無駄にすることは森林資源上好ましくないことは十分承知してはいるのですが、いつでもどこでもデバッグできるのは魅力です。紙を単なる紙だと思わずに、表示装置として考えてみるとそれも納得できるでしょう。紙なら電源も必要ありませんし、好きなところにすぐアクセスできて赤でいろいろ記入できるのです。これはパソコンの画面がかなり大きくないと同じようなことができないので、当分の間は紙に印刷したプログラムリストを眺める時代が続きそうです。

そこで、ここではプリンタ制御も含めトレーニングするため新たにオリジナルのリスターを開発しましょう。中身が自分の作ったものならば、どこをどう改造すればよいのか簡単にできるし、必要な機能だけに絞ってコンパクトなリスターが作れます。

■リスト 11-1

```
 1: program LIST11_1;
 2: uses Crt;
 3:   var
 4:     inbuffer : string[135] ;
 5:     fiLename : string[14] ;
 6:     ListfiLe , infiLe : text ;
 7:     pagecount,Linecount : integer ;
 8:     printer : booLean ;
 9:
10:   const
11:     MAXLINE = 52 ;
12:     FORM_FEED = #12 ;
13: {---------------------------------------------------------}
14:
15:   procedure Input_FiLe;
16:
17:     var
18:       existing : booLean ;
19:
20:     begin
21:       repeat
22:         Write(' Enter FiLe Name ',
23:               'to List or <CR> to Exit ');
24:         ReadLn(fiLename);
25:           if fiLename = '' then haLt;
26:           if Pos('.' , fiLename) = 0 then
27:             fiLename := Concat(fiLename , '.pas');
28:           Assign(infiLe , fiLename);
29:               {$i-}
30:           Reset(infiLe);
31:               {$i+}
```

```
32:        existing := (IOResuLt = 0);
33:        if not existing then
34:          begin
35:            WriteLn(' FiLe Doesn''t Exist');
36:            DeLay(700);
37:          end;
38:    untiL existing;
39:  end;
40: {-----------------------------------------------------}
41:
42:  procedure Out_Dev;
43:   var
44:     c : char ;
45:   begin
46:     repeat
47:       Write(' Output Listing to ',
48:             '(C)onsoLe or (P)rinter ?  ');
49:       ReadLn(c);
50:       c := UpCase(c);
51:     untiL c in ['C','P'];
52:     WriteLn;
53:     if c = 'C' then
54:       begin
55:         Assign (ListfiLe , 'CON');
56:         printer := FALSE;
57:       end
58:     eLse
59:       begin
60:         Assign (ListfiLe , 'PRN');
61:         printer := TRUE;
62:       end;
63:     ReWrite(ListfiLe);
64:   end;
65: {-----------------------------------------------------}
66:
67:  procedure Line_Print;
68:   begin
69:     if Linecount<10 then
70:       WriteLn(ListfiLe,'   ',Linecount,': ',inbuffer)
71:       eLse
72:       if Linecount<100 then
73:         WriteLn(ListfiLe,'  ',Linecount,': ',inbuffer)
74:         eLse
75:         if Linecount<1000 then
76:           WriteLn(ListfiLe,' ',Linecount,': ',
77:                                         inbuffer)
78:         eLse
79:         WriteLn(ListfiLe,Linecount,': ',inbuffer);
80:         end;
```

```
 81: {-----------------------------------------------------------}
 82:
 83:   procedure Skip_Page;
 84:     begin
 85:       if printer then
 86:         Write(ListfiLe,FORM_FEED)
 87:         eLse WriteLn;
 88:       WriteLn(ListfiLe,'TURBO PASCAL SOURCE CODE',
 89:                ' LISTER : Listing of ',fiLename,
 90:                       '       PAGE   ',pagecount);
 91:       WriteLn(ListfiLe);WriteLn(ListfiLe);
 92:       pagecount := Succ(pagecount);
 93:     end;
 94:
 95: {===========================================================}
 96:
 97:   begin
 98:     CLrScr;
 99:     repeat
100:       Input_FiLe;
101:       Out_Dev;
102:       Linecount := 1;
103:       pagecount := 1;
104:       Skip_Page;
105:
106:       whiLe not eof(infiLe) do
107:         begin
108:           ReadLn(infiLe,inbuffer);
109:           Line_Print;
110:           Linecount := Succ(Linecount);
111:           if (Linecount>0)
112:                 and (Linecount mod MAXLINE=0) then
113:             Skip_Page;
114:         end; {whiLe not eof}
115:       if printer then Write(ListfiLe,FORM_FEED);
116:     untiL FALSE
117:   end.
```

リスト 11-1 を見てください。これがリスターです。このリスト自体リスト 11-1 で作ったものです。

リスター設計上必要となるのは、リストをとるべきソースファイル名の入力、出力装置の選択、用紙1ページに印刷する行数でしょう。これをすべて手続きにしてしまえば、プログラムはかなり簡単に作れます。TURBO Pascal をトレーニングしてきた読者なら、当然のプログラム製作法ですね。

まず、15 ～ 39 行の手続き、Input_File でファイル名の入力を行います。何も入力せ

ずにリターンだけ入れるとプログラムは停止します。また、TURBO Pascal 専用のリスターにしますから、拡張子を省略したら自動的に.PAS が付加されるようにしておきます。さらに、存在しないファイルのリストをとろうとしたときには、いきなりリスターが停止するような「失礼なこと」をせず、再入力できるようにしておきます。この手続きはもう何度も使っているので解説の必要はないでしょう。

42 ～ 64 行の手続き Out_Dev は出力用のファイル名に CON を指定するのか PRN を指定するのかを決めるものです。67 ～ 80 行の Line_Print は印刷する行が何行目かの桁数でずれたりしないように調整するための手続きです。ここをもう少しきれいにつくるのなら、write パラメータを利用するとよいでしょう。

83 ～ 93 行の Skip_Page は印刷最大行数 MAXLINE まで用紙 1 ページに印刷したときの処理です。出力ファイルがプリンタであれば用紙送りのコードを出力します。このコード (特殊文字) は大体どのメーカーのプリンタでも同じですが、特殊な用紙送りを実現したければ定数 FORM_FEED を変えてやればできます。

97 行目から始まるメインプログラムも説明の必要はなさそうです。ただ、99 行目の repeat に対応する 116 行目の until の条件は絶対に成立しません。つまり、この間は無限ループになっています。このループから抜けるには、ファイル名入力時にリターンだけを押下して 25 行目で Halt になる場合だけです。ついでに注意しておきますが、36 行目の Delay がないと 35 行目のメッセージは一瞬しか表示されないので見えません。あと注意すべき点は、メインプログラムの 112 行目で 52 行印刷するたびに改ページしていることくらいです。

このリスターを改造するとしたらどこでしょうか。簡単なものではページの頭にある表題 (ヘッディング) をあなたの専用文字にすることです。もう少し難しくするのなら、長い行の折り返しをきれいに処理することがあるでしょう。もっとやってみたければ、Pascal の予約語は強調印字されたり、そこだけ字体が変わるようなリスターも考えられるでしょう。実用性では L の小文字と数字の 1 をプリンタ上ではっきり区別するために、L の小文字を外字でわかりやすいフォントに作ることも考えられます。つまり、L の小文字をすべてそれに置き換えて印刷するのです。これはプリンタのマニュアルでエスケープ・コードを使用した外字印刷のところを参考にすればできるはずです。

プログラムの関数・手続きを把握する

11–2■サブプログラムを一覧させる

TURBO Pascal でプログラム開発をしていると、いろいろなプログラム・ファイルがたまってきます。それはそれでよいのですが、本数が多くなるといったいどこにどんな手続きや関数があるのかわからなくなります。せっかく苦労して作った便利な手続きや関数がどのファイルに入っているのかわからず、記憶を頼りに再び同じプログラムを

作る羽目になってしまいます。これでは「知的財産」の管理ができていないということになります。

そこで、その管理をパーソナルコンピュータにまかせるため、TURBO Pascalでプログラムファイルの中にある関数と手続き、つまりサブプログラムを一覧表にするプログラムを開発してみましょう。プログラムの名前と関数と手続きの名前、およびその引数を一覧表にして印刷し、ファイルにとじておけば、どこに何があるのかすぐにわかります。ついでにメモも記入して、すぐに思い出せるようにしておけば万全です。

このようなプログラムを作る方針としてはプログラム・ファイルを読み込んで、その中からprogram、procedure、functionの綴りを探し出すということになります。しかし、これら綴りの大文字小文字にとらわれないで探す必要があります。たとえば、Programと書いてあってもpRoGrAmと書いてあっても見つけねばなりません。これは一見難しそうですが、TURBO Pascalに備えられている文字関数UpCaseを利用すれば簡単にできます。この関数で、読み込文字列を次々にすべて大文字に変換してPROGRAMを探すようにすればよいのです。この点に気がつきさえすればそれほど面倒なことにはならないでしょう。

ほかの部分は、いままでのトレーニングを応用すれば何とかなりどうな気がすると思います。すでにTURBO Pascalで長いプログラムをどうやって開発するかを身に付けたひとなら、いきなりエディタに向かってプログラムを作り出せるでしょう。とにかく必要と思われる手続きから作り始めるのがコツです。こうして作ったのがリスト11-2のプログラムです。

■リスト 11-2

```
 1: program LIST11_2;
 2: uses Crt;
 3:
 4:  type
 5:    Str14   = string[14];{x:yyyyyyyy.zzz}
 6:    Str255       = string[255];
 7:    fiLe_type  = text;
 8:
 9:  var
10:    in_fiLe_name  : Str255;
11:    input_fiLe  : fiLe_type;
12:    List_fiLe : text;
13:    Line_buff,
14:    first_word,
15:    SPword            : Str255;
16:    StChar      : string[1];
17:    printer          : booLean;
18:    count        : integer;
19: {------------------------------------------------------}
20:
```

```
21: procedure Get_FiLeName;
22:
23:   const
24:     BELL      = #7;
25:
26:   var
27:     Check        : booLean;
28: {--------------------------------------------------------}
29:
30:   procedure Printer_OK;
31:
32:    var
33:     Character    : char;
34:
35:    begin
36:      printer:=faLse;
37:      Write('手続き・関数リストを印刷しますか？[ Y to yes ] ');
38:      ReadLn(Character);
39:      if (Character='Y') or
40:         (Character='y') or
41:         (Character='ン')
42:      then
43:        begin
44:          Assign(List_fiLe,'PRN');
45:          ReWrite(List_fiLe);
46:          printer:=true;
47:        end;
48:    end;
49: {--------------------------------------------------------}
50:
51:   procedure FiLe_Check;
52:
53:    begin
54:      if Pos('.' , in_fiLe_name) = 0 then
55:           in_fiLe_name := Concat(in_fiLe_name , '.pas');
56:      {$I-}
57:      Assign(input_fiLe,in_fiLe_name);
58:      Reset(input_fiLe);
59:      {$I+}
60:      Check:=IOResuLt=0;
61:    end;
62: {========================================================}
63:
64:   begin
65:     Check:=faLse;
66:     WriteLn(' ');
67:
68:     repeat
69:       in_fiLe_name:='';
```

```
 70:      WriteLn;
 71:      WriteLn;
 72:      Write('リストを取りたいファイル名を指定してください。: ');
 73:      ReadLn(in_fiLe_name);
 74:      if (in_fiLe_name='') then HaLt
 75:      eLse
 76:        begin
 77:          FiLe_Check;
 78:          if not Check
 79:          then
 80:            WriteLn('そのファイルは存在しません。',BELL)
 81:          eLse
 82:            begin
 83:              Printer_OK;
 84:            end;
 85:        end;
 86:    untiL Check;
 87:
 88:  end;
 89:
 90: {--------------------}
 91:
 92: procedure Print_TitLe;
 93:
 94:  begin
 95:    WriteLn(' ');
 96:    WriteLn('********    LIST    ********');
 97:    WriteLn(' ');
 98:    WriteLn(in_fiLe_name,'の手続き・関数');
 99:    WriteLn(' ');
100:    if printer
101:    then
102:      begin
103:        WriteLn(List_fiLe,' ');
104:        WriteLn(List_fiLe,'********    LIST    ********');
105:        WriteLn(List_fiLe,' ');
106:        WriteLn(List_fiLe,in_fiLe_name,'の手続き・関数');
107:        WriteLn(List_fiLe,' ');
108:      end;
109:  end;
110:
111: {--------------------}
112:
113: function Char_UpCase : Str255;
114:
115:  var
116:    UpWord      : Str255;
117:     I          : integer;
118:
```

```
119:  begin
120:    UpWord:='';
121:    for I:=1 to Length(first_word) do
122:      begin
123:        StChar := Copy(first_word,I,1);
124:        UpWord:=Concat(UpWord,UpCase(StChar[1]));
125:      end;
126:    Char_UpCase:=UpWord;
127:  end;
128:
129: {-----------------}
130:
131: procedure Get_FirstWord;
132:
133:  var
134:    Pass       : booLean;
135:    Character  : char;
136:
137:  begin
138:    first_word:='';
139:    count:=1;
140:    Pass:=true;
141:
142:    whiLe (Pass) and ( count<=Length(Line_buff) ) do
143:      begin
144:        StChar := Copy(Line_buff,count,1);
145:        Character:= StChar[1];
146:        if ( (Character>='0') and (Character<='9') ) or
147:           ( (Character>='A') and (Character<='Z') ) or
148:           ( (Character>='a') and (Character<='z') ) or
149:           ( Character='_') or ( Character='{' )
150:        then
151:          first_word:=Concat(first_word,Character)
152:        eLse
153:          begin
154:            if (Length(first_word)>0) or (Character='{')
155:            then Pass:=faLse;
156:          end;
157:        count:=count+1;
158:      end;
159:
160:  end;
161:
162:  {-------------------}
163:
164: procedure Get_PF_Name;
165:
166:  var
167:    Character   : char;
```

```
168:
169:  begin
170:
171:     repeat
172:       StChar := Copy(Line_buff,count,1);
173:       Character:= StChar[1];
174:       if Character='('
175:       then
176:         begin
177:
178:           repeat
179:             Write(Character);
180:             if printer
181:             then Write(List_fiLe,Character);
182:               count:=count+1;
183:               if count>Length(Line_buff)
184:               then
185:                 begin
186:                   Write('');
187:                   if printer
188:                   then Write(List_fiLe,'');
189:                   ReadLn(input_fiLe,Line_buff);
190:                   count:=1;
191:                 end;
192:               StChar := Copy(Line_buff,count,1);
193:               Character:= StChar[1];
194:           untiL Character=')';
195:
196:         end;
197:       Write(Character);
198:       if printer
199:       then Write(List_fiLe,Character);
200:       count:=count+1;
201:       if count>Length(Line_buff)
202:       then
203:         begin
204:           Write('');
205:           if printer
206:           then Write(List_fiLe,'');
207:           ReadLn(input_fiLe,Line_buff);
208:           count:=1;
209:         end;
210:     untiL Character=';';
211:
212:  end;
213:
214:    {==================}
215:
216:  begin
```

```
217:   CLrScr;
218:    Get_FiLeName;
219:
220:    repeat
221:        Print_TitLe;
222:
223:        repeat
224:          ReadLn(input_fiLe,Line_buff);
225:          Get_FirstWord;
226:          SPword:=Char_UpCase;
227:          if (SPword='PROGRAM')    or
228:             (SPword='PROCEDURE') or
229:             (SPword='FUNCTION')
230:          then
231:            begin
232:              WriteLn('');
233:              Write(first_word,' = ');
234:              if printer
235:              then
236:                begin
237:                  WriteLn(List_fiLe,'');
238:                  Write(List_fiLe,first_word,' =');
239:                end;
240:              Get_PF_Name;
241:            end;
242:        untiL eof(input_fiLe);
243:
244:        CLose(input_fiLe);
245:        Get_FiLeName;
246:
247:   untiL TRUE = FALSE;
248:  end.
```

まず一覧表を作る対象となるプログラム・ファイルの名前とそれが存在するかどうかを調べなければなりません。そこで、21 行目のような Get_FileName という手続きから まず作り始めます。もしもファイルが存在しないときには警告ブザーを鳴らしたいので、定数として BELL を定義しておきます。そして、存在するときには TRUE になる論理変数 Check も定義しておきます。

ここまできたところで、リストをプリンタと画面かのいずれに出力すべきかを決めておかねばならないことに気がつきましたので、30 行目で手続き Printer_OK の記述に入ります。48 行目までは出力装置としてプリンタを指定するかどうかだけですから、難しいところは何もないでしょう。46 行目で printer を TRUE にしておきます。

ここで今度はファイルの存在の有無を調べる手続きが必要なことに気がつきました。そこで 51 行目のようにファイルの存在の判定をする手続き File_Check に取りかかりま

す。この手続きではプログラム・ファイルの拡張子が省略されたときには自動的に.pas を付加するようにしています。64 行目から Get_FileName 本体の作成が始まります。88 行目までのプログラムはわかりやすいので、行を追うだけで十分理解できるでしょう。

92 行目の手続き Print_Title は一覧表のヘッディング用のものです。内容は画面表示してプリンタに出力してよいのなら同じものを印刷します。ここまでくればいよいよファイルの中から program 、procedure 、function を探す手続きを作る段階になります。その前に上で述べたすべての文字列を大文字に変える手続きを作っておきましょう。

113 行目の手続き Char_UpCase は文字列 first_word をすべて大文字に置き換えます。標準関数 UpCase は 1 文字を大文字に変えるだけですから、これを繰り返し使って文字列全体を大文字に置き換えるのです。この手続きの内容もとても簡単です。121 行目の for 文で first_word の中に含まれる英字を 1 個ずつ大文字にしています。

program などはいずれにしても Pascal 文の先頭にあるはずですから、各行の先頭の単語を拾い出す手続き Get_FirstWord を 131 行目から作ります。142 ～ 158 行目で行の先頭から文字が 0 ～ 9、A ～ Z、a ～ z かあるいは_(アンダースコア) のように有効な名前を構成するものである限り first_word に拾っていきます。この while 文を抜けたときには Pascal の単語が first_word の中に入っているというわけです。

最後に残った「難物」の引数を出力する手続き Get_PF_Name を 164 行目から作ります。171 ～ 194 行目で引数の並びがおわる右括弧を見つけるまで ( ) の間にある引数を次々に出力します。

ここまで手続きができてしまえば、あとはサブプログラム名の出力だけですから、216 行目からメインプログラムを書き始めます。まず、217 、8 行目で画面をクリアしてから一覧表を作りたいプログラムファイル名を入力します。220 ～ 247 行目の間は TRUE が FALSE になるまで続けるので、無限に繰り返します。

223 ～ 242 行目の繰り返しは入力されたプログラム・ファイルの EOF(エンド・オブ・ファイル) マーク、つまり^Z を読み込むまでに PROGRAM 、PROCEDURE 、FUNCTION というキーワードが行の先頭にあるたびに、それを一覧表として 231 ～ 241 行目で出力します。

すべての行の処理がおわると 244 行目でプログラム・ファイルをクローズし、245 行目で再び次に一覧表を作りたいプログラム・ファイル名を入力して同じ処理を続行します。こうしてみると、247 行目の条件は絶対に成立しないので無限に繰り返すことになりますが、実は Get_FileName 手続きの中の 74 行目で入力ファイルを指定せずリターンキーだけを押下すると Halt でプログラム実行を終了します。

COMMAND.COMを参照する

11–3■環境変数の参照

MS-DOS でよく「環境」という言葉が使われますが、実際には聞いたことがあってもどう使ったらよいのかわからないのではありませんか。ここではそれをどうすれば活用できるのかプログラムを作ってみることにします。TURBO Pascal ver5.0 以降では EnvCount、EnvStr などの関数が用意されています。しかし、ここではわざわざそれを使用しないで TURBO Pascal でプログラミングしています。

これらの関数をそのまま利用できるのであればわざわざこんなことは必要ないのですが、引数を標準のものから変えたかったり、ちょっと変更したいときには環境変数の参照をどうすればよいのかが必要になります。ここで紹介するプログラムはあくまでも環境変数の参照の出発点ですから、ぜひいろいろな改造にチャレンジしてみてください。

このプログラムは、簡単にいうと現在使用しているコマンドプロセッサ名（普通は COMMAND.COM です）を表示した後、そのコマンドプロセッサを起動してディレクトリの簡易表示である “DIR /W” コマンドを実行します。そしてこの後、環境変数 “PATH” の内容を表示します。

■リスト 11-3

```
 1: program LIST11_3;
 2: {$M 16384, 0, 0}
 3:
 4: uses  dos, LIST11_4;
 5:
 6: var
 7:   myst : string;
 8:
 9: begin
10:   { コマンドプロセッサを探す }
11:   myst := 'COMSPEC';
12:   WriteLn('The command processor is: ', getenv(myst));
13:
14:   { シェルに抜けてｄｉｒを実行 }
15:   SwapVectors;
16:   Exec(getenv(myst),'/C DIR /W');
17:   SwapVectors;
18:   WriteLn('DosError is( no error is 0 ):',DosError);
19:
20:   { コマンドのパスを表示 }
21:   myst := 'PATH';
22:   WriteLn('The command path is: ', getenv(myst));
23: end.
```

リスト 11-3 を見てください。このリストは後で説明するリスト 11-4 のユニットを使

用しています。11、12行目で現在使用しているコマンドプロセッサを表示しています。MS-DOS 起動時に CONFIG.SYS ファイルで SHELL を指定していなければ、ここで COMMAND.COM が表示されるはずです。

15、17行目は EXEC 手続きを使うときの定石です。SwapVectors 手続きは Dos ユニットに用意されている手続きで、これを実行しておかないと Exec 手続きで起動されたプロセスが現在の割り込みベクタを破壊して暴走する可能性があります。さらに2行目のコンパイラ指令でスタック領域を確保しています。Exec 手続きは起動される子プロセスによってはメモリが不足する可能性があるからです。ここでは 16KB をスタックとして確保し、ヒープ領域はまったく確保していません。

16行目で COMMAND.COM を子プロセスとして起動し、"DIR /W" を実行しています。"/C" は COMMAND.COM から制御を取り戻すために必要です。これがないと "EXIT" と入力されるまでコマンドプロセッサが実行されたままになります。もちろん、そうしたいこともよくありますから、その場合には/C を省略します。なお、このパラメータは COMMAND.COM 以外のアプリケーションには必要ありません。

18行目は EXEC 手続きを実行した際のエラーを表示しています。本来、手続きは何も値を返さないのですが、EXEC 手続きは MS-DOS のファンクションなので実行すると MS-DOS が値を返してきます。つまり、エラーコードを返してくるわけです。このエラーコードは各手続きや関数毎に返すのではなく、"DosError" という DOS ユニットに定義された変数に返されます。21、22行目で現在の "PATH" を表示しています。

ざっと眺めただけでは難しいことは何もないはずです。12、16、22行目にある Getenv 関数がこのプログラムで中心的役割をしているので、この関数が定義されているユニット LIST11-4.PAS を説明しましょう。

■リスト 11-4

```
 1: unit  LIST11_4;
 2:
 3: interface
 4:
 5: uses  dos;
 6:
 7: function GetEnv(st : string) : string;
 8:
 9: impLementation
10:
11: function GetEnv(st : string) : string;
12: const
13:    PrefixSegOfs = $2c; { ＰＳＰ内の環境セグメントアドレスのオフセット }
14: var
15:   tmpst  : string;
16:   envseg : word;
17:   i      : integer;
```

```
18:
19: begin
20:   envseg := memw[PrefixSeg: PrefixSegOfs]; { ＰＳＰからセグメント取得 }
21:
22:   for i := 1 to Length(st) do
23:     st[i] := UpCase(st[i]);  { 大文字に変換 }
24:
25:   i := 0;
26:   whiLe (mem[envseg : i] <> 0) do { 環境文字列の処理 }
27:    begin
28:      tmpst :='';
29:      GetEnv := '';
30:
31:      whiLe (mem[envseg : i] <> 0) do
32:       begin
33:         tmpst := tmpst + UpCase(Chr(mem[envseg :i])); { 文字の結合 }
34:         Inc(i);
35:       end;
36:
37:       if (Pos( st, tmpst) = 1) then
38:         GetEnv := Copy(tmpst, Pos('=',tmpst)+1, 255)
39:       eLse Inc(i);
40:
41:    end;
42: end; { GetEnv }
43:
44: end.
```

1〜9行目まではユニットの定石ですからすぐに理解できると思います。11行からがGetenv関数の本体です。この関数はPSP(Program Segment Prefix)内の環境変数エリアの(セグメント)アドレスを参照して内容を読みとろうというものです。

まず、20行目でPSPから環境変数が保存されているセグメントアドレスをenvsegという変数にセーブしておきます。22、23行目で引き渡された文字列(参照したい環境変数)を大文字に変換します。25行目以降で目的の環境変数を探しています。31〜35行目で1つの環境変数を探して大文字に変換します。

見つかった環境変数が目的の環境変数かどうかを判定しているのが37行目です。ここで、目的の環境変数であることが判明すると次の38行目でこの関数のリターン値として見つかった環境変数をセットしています。目的のものでなかった場合には39行目でポインタiを次に進めています。

26行目と31行目のWHILE文に条件0を使うのは理由があります。環境変数に限らず、MS-DOSで使用する文字列はASCIZ文字列を使っているためです。ASCIZでは以前にも解説したように、ASCII文字の集合である文字列の終りを認識するのにASCIIコードの0(NULL)を付加して行っているからです。

TURBO Pascal では文字列の長さを認識するために、文字列の長さを表す 1 バイトを先頭につけて管理していますが、ASCIZ では異なっているので注意が必要です。TURBO Pascal の文字列管理方法では、長さを表すバイト数が 1 バイトなので扱える長さには制限がありますが、ASCIZ 文字列では制限がほとんどありません。ここで「まったく」ではなく「ほとんど」としたのは、管理方法としては制限を受けないのですが、セグメントなどハードウェア的に制限を受けてしまうためです。

この節の最初で断っておきましたが、ここで紹介した GetEnv 関数は TURBO Pascal の Dos ユニットの中に標準で用意されています。しかし、標準で用意されている手続きの中でいったいどうやってマニュアルに記載してある機能を実現しているのかは外からではわかりません。もちろん、逆アセンブルまでやるほどの根性 (?) を入れれば別ですが、既存の手続きや知識を総合すれば同じものが簡単に実現できることがおわかりいただけたでしょう。

ハードコピーツールの作成

11-4■グラフィック画面の印刷

TURBO Pascalver5.0 以上では標準で BGI が用意され、簡単にグラフィックスが表示できるようになりました。今までは自作のライブラリか BASIC の ROM ルーチンを利用するなどめんどうなことだったのですが、一挙に楽になったのです。

しかし、せっかく美しい画面表示のプログラムを作成しても、この表示を印刷する機能が標準では用意されていないのは片手落ちです。TURBO Pascal を日本語化した MSA 社では、このことに気付いていて BGIUTILS.ARC という圧縮ファイルの中にアセンブラで記述したプログラムを付けてくれています (Thanks MSA！)。

■リスト 11-5

```
 1: unit  LIST11_5;
 2:
 3: interface
 4:
 5: Uses  printer, Graph;
 6:
 7: Procedure GraphPrint(x1, y1, x2, y2 : integer);
 8:
 9: impLementation
10:
11: Uses    Dos;
12:
13: procedure OutputByte(SendData : byte);
14: { プリンタへの１バイト出力 }
15: var
```

```
16:   reg : registers;
17:
18: begin {OutputByte}
19:   repeat
20:     reg.ah := $12;
21:     Intr($1a, reg);
22:   untiL (reg.ah and 1) = 1;{ データ出力可能まで }
23:
24:   reg.ah := $11;
25:   reg.aL := SendData;
26:   Intr($1a, reg);
27: end; {OutputByte}
28:
29:
30: Procedure GraphPrint(x1, y1, x2, y2 : integer);
31: var
32:   BackGroundCoLor,
33:   MaxWidth,
34:   MaxLength : word;
35:   MaxWidthString : string[4];
36:
37:   PrintPattern : byte;
38:   count,
39:   Row, CoLumn, PixeL : integer;
40:
41:   VPT : ViewPortType;
42:
43: begin
44:   BackGroundCoLor := GetBkCoLor; { バックグラウンドカラーをセーブ }
45:   Maxwidth := x2 -x1;            { 有効幅 }
46:   MaxLength := (y2 - y1) div 8;  { 有効長 }
47:
48:   str(MaxWidth+1 :04 ,MaxWidthString); { 有効幅を文字に変換 }
49:   for count := 1 to 4 do               { スペースを0に変換 }
50:     if MaxWidthString[count] = ' ' then
51:       MaxWidthString[count] := '0';
52:
53:   GetViewSettings(vpt);                  { ビューポートをセーブ }
54:   SetViewPort(0,0,GetMaxX,GetMaxY,faLse); { ビューポートをセット }
55:
56:   Write(LST, Chr(27),'T12'); { 改行モードを 12/120 インチにセット }
57:   Write(LST, Chr(27),'D');   { コピーモードにセット }
58:
59:   for row := 0 to MaxLength do
60:     begin
61:       Write(LST, Chr(27),'S',MaxWidthString);{8 ドット列対応グラフィック}
62:
63:       for coLumn := x1 to x2 do   { 横1列分のイメージを作成 }
64:         begin
```

```
65:       PrintPattern := 0;
66:       for pixeL := 0 to 7 do  { 縦 8 ドット分のデータを作成 }
67:         if ((y1 + row *8 + pixeL) <= y2) and
68:            GetPixeL(coLumn, y1 +row *8 +pixeL) <> BackGroundCoLor) then
69:         case  pixeL of { 画面とヘッドとを対応させる }
70:            0 : PrintPattern := PrintPattern + $01;
71:            1 : PrintPattern := PrintPattern + $02;
72:            2 : PrintPattern := PrintPattern + $04;
73:            3 : PrintPattern := PrintPattern + $08;
74:            4 : PrintPattern := PrintPattern + $10;
75:            5 : PrintPattern := PrintPattern + $20;
76:            6 : PrintPattern := PrintPattern + $40;
77:            7 : PrintPattern := PrintPattern + $80;
78:          end; { of case }
79:       OutPutByte(PrintPattern);
80:     end; { of for }
81:    WriteLn(LST);
82:   end; { of for }
83:
84:   WriteLn(LST, Chr(27),'A'); { 1/6インチ改行モードセット }
85:   Write(LST, Chr(27),'M');    { ネイティブモードセット }
86:   SetViewPort(VPT.x1,VPT.y1,VPT1.x2,VPT.y2,VPT.cLip);
87:   {ビューポートリセット}
88: end;{ of graphprint }
89:
90: end.{ of unit }
```

MSA 社ではこのプログラムをユーザーが組み込んで自由に使用してもかまわないといっていますが、TURBO Pascal しか知らないユーザーがこのようなアセンブラで記述されたプログラムを使うことは大変難しいといえます。それにちょっと変えてみようかと思ってもアセンブラの知識がないとどうしてよいのかわかりません。

ちょっと腕のある(うるさい?)ひとならアセンブラなどなんとも思わず楽々プログラミングできるのでしょうが、後で改造したりするときのためにも高級言語で記述できるものならそうしたいところです。

そこで、ここではTURBO Pascal でグラフィック画面を印刷するプログラムを作ってみようというのです。

リスト 11-5 では他のプログラムに簡単に組み込むことができるようにユニットとして記述しました。ハードコピーを行なう手続きが GraphPrint 手続きです。この手続きは印刷する座標を指定することができるため、様々な使い道があると思います。

■リスト 11-6

```
  1: program LIST11_6;
  2:
  3: Uses  graph, LIST11_5;
```

```
 4:
 5: var
 6:   graphdriver, graphmode, radius : integer;
 7:
 8: begin
 9:   graphdriver := detect;                 { 自動検出したドライバ }
10:   InitGraph(graphdriver, graphmode,''); { グラフィック画面の初期化 }
11:
12:   RectangLe(0,0,GetMaxX, GetMaxY);       { 枠をかく }
13:
14:   for radius := 1 to 10 do  { 中心から円をかく }
15:     arc(GetMaxX div 2, GetMaxY div 2, 0, 360, radius *(GetMaxY div 10));
16:
17:   OutTextXY(10, 10, 'This is HorizontaL Text to heLp test GraphPrint');
18:
19:   SetTextStyLe(sansSerifFont,vertdir,2);{ 下から上へ }
20:   OutTextXY(50,50,'VerticaL Text');
21:
22:   Graphprint(0,0,GetMaxX, GetMaxY); { ハードコピーをとる }
23:
24:   CLoseGraph;
25: end.
```

では、プログラムの内容を見てみましょう。9行目まではユニットの定石であるお決まりの記述です。13行～27行目はプリンタに1バイトを出力するための手続きです。プリンタに文字を出力するにはPascalの標準出力手続きであるWriteLnを使用して、出力装置にプリンタを指定するのが普通です。つまり、

WriteLn(LST,*****)；

のようにするのが一般的なのです。わざわざこの手続きを作ったのには理由があります。私も初めはこのようにPascalのWriteLn文を使用してプリンタにデータを出力していたのですが、何度試してもうまく出力されなかったのです。始めのうちはプログラムのバグだと思って他の部分をデバッグしていたのですが、デバッグの結果他の部分は間違っていないということが判明しました。そこで、思い切ってプリンタBIOSを使ってこの手続きを作り、実行してみたところうまくいったので、このようになったというわけです。このように自分のプログラム以外の原因でプログラムが正常に動作しないときはホトホト困ってしまいます。

さて、グチはこのくらいにして説明を続けることにします。30行目からがGraphPrintの本体にあたります。このプログラムはBGIを使っているのでプリンタ関係の所だけを修正すれば他のマシンにも対応できるはずです。また、ターゲットプリンタはNECのPRシリーズです。これも制御コードを変更することで他のプリンタを使用することが可能です。

プリンタを購入すると立派な分厚いマニュアルがついてきます。しかし、標準的な使用しかしないユーザーは最初の「パソコン機種別接続法」あたりを読むだけで、後の大部分は読まないでしょう。しかし、ここで本棚に投げ込んでおいたプリンタのマニュアルががぜん活躍します。もしも、このユニットをNECのPRシリーズ以外に移植したいのなら、マニュアルの制御コードのところを参照する必要があります。機種によっては、グラフィックモードを持っていないものもあるので注意してください。

さて、44行目で現在のバックグラウンドカラーをセーブしています。45、46行目で引数として渡された座標を計算して、有効な幅や長さを求めます。49～51行目で有効幅を文字列に変換します。これはプリンタに送る制御コードが文字の形式だったために、このように面倒な変換をせざるをなりません。この変換では始めに有効幅をStr手続きで数字から文字に変換します。このままではプリンタへ送る制御コードの形式にあっていないので、空白(スペースコード)をさがして0に変換します。

53行目で現在のビューポートをセーブし、54行目で新しいビューポートをセットします。56、57行目でプリンタに制御コードを送ります。59～82行目でプリンタをグラフィックモードにする制御コードを送り、画面のデータをプリンタのデータに変換して印刷します。61行目でプリンタをグラフィックモードにセットしています。63～82行目で1行分の印刷を行なっています。65行目でこれから印刷する変数を初期化します。67～78行目は画面データをプリンタヘッドのイメージに変換する部分です。この変換のメインとなるのが69～78行目までのcase文です。

79行目が先ほどの苦労したプリンタへのデータ転送の部分です。84、85行目はプリンタを標準のモードに戻しておくために必要です。86行目でセーブしておいたビューポートを元に戻します。このような後始末をしておかないと、このユニットを利用しているプログラムの中で他にもプリンタ出力があるとおかしなことがおこりかねません。

このユニットを試してみるために作ったのが、リスト 11-6 です。このデモプログラムは枠を描いて、その中に円を描いた画面をハードコピーする単純きわまりないものですが、一応このプログラムの内容も説明しておきましょう。

9、10 行目は BGI を使うときの定石ですから必ずこのようにしてください。12 行目で画面に枠を描いています。14、15 行目で画面に同心円を描きます。17 ~ 20 行目で今度は文字を書いてみます。17 行目で普通に (10,10) の座標から文字を書いているのですが、20 行目で文字を書くときにはちょっと細工をしてみました。19 行目がその細工です。この手続きで下から上に表示するようにしてあるのです。ここまでで画面に表示するのをやめて、そろそろハードコピーをとってみたくなったので、次の 22 行でハードコピーをとることにしました。24 行は BGI を使い終わったときに実行する定石ですから、先ほどの 9、10 行目とこれをペアにして忘れずに必ず実行しておくようにしてください。

これで、ハードコピーがとれるプログラムを作るのがすごく楽になったと思いますし、CG(コンピュータグラフィックス) をやる人には欲しかったツールでしょう。このプログラムを生かすも殺すもあなたが描く画面表示次第です。ハードコピーとして残しておくのにふさわしい芸術作品 (?) をどんどん作って紙に印刷してみましょう。

IPLプログラムを改造する

11–5■データディスクのブートメッセージ表示

パソコンを立ち上げるときに、システムディスクと間違えて、データディスクをフロッピーディスクドライブに入れてしまったことはだれにもある経験です。このときの“ピー”というやかましい音はいやなものです。この音が鳴出すとキー入力も受け付けなくなります。

この音を止めるにはリセットキーを押すしかありません。夜中にパソコンを使っているときに (パソコンマニアが本格的に活動するのはぼくも含めて大抵この時間ですよね！) この音が鳴って、周りに迷惑をかけるのは困ったものです。それでなくても役に立つのか立たないのかわからないものに多額の投資 (ムダ遣い?) をしているぼくらマニアはいつも家族に白い目で見られているだけに、なんとかこの音をやめさせたいと思ったわけです。

フロッピーディスクからパソコンを起動すると、どんな順序でものごとが進行するのでしょうか。ここから考えてみます。

電源が投入されるかリセットボタンが押されると、まずパソコン内部のメモリが初期化されます。そして、そのときに挿入されているフロッピーディスク上の特定の場所からデータがプログラムとして読み込まれます。次にそのプログラムに制御を移して本格的にパソコンが立ち上がるという手順が自動的に行われます。

この手順の中で、「フロッピーディスク上の特定の場所」から読み込んだプログラム

をパソコンが解析したら、システムとして実行できないものなので、あのやかましい音を出して警告していたのです。このプログラムは現在の番地にジャンプするという無限ジャンプも行なっていたので、キー入力もできなくなるのです。

そこで、この立ち上げ時に読み込まれるプログラムを何とかしてしまえば音を出さないように変えられるのではないかと考えられます。つまり、このプログラムをそっくり入れ換えて、もっと使いやすく、しかもやかましくないものに改造しようというわけです。

具体的には、立ち上げ時にシステムディスク以外が挿入されているときには指定されたメッセージを表示して、キー入力を待つようにします。そして、何かキー入力があればリセットがかかり、再起動を行なってくれればよさそうです。このキー入力をする前に正しいフロッピーディスクに差し替えておけば、そこでパソコンが正しく立ち上げられるというわけです。

むろん、またまた間違えてシステムディスク以外のものを挿入してしまうとリセットしなくてはならないのは当然です。ここで表示されるメッセージはプログラムと一緒にフロッピーディスクに書き込んでおきます。ただ音を出さずにじっと待っているのでは何が原因で立ち上がらないかわからないからです。そのためのメッセージはファイルとして記録しおき、そのファイル名を指定する形で表示メッセージの内容を決めています。こうしておけば自分の好みのメッセージに簡単に変えられるからです。

MS-DOSではシステムディスク以外のフロッピーディスクにはどのようなプログラムが書いてあるのかをまず調べてみましょう。MS-DOSに付属してくるSYMDEBを利用してその部分を解析した結果の逆アセンブルリストの一部分をここで示します。説明上の混乱を避けるためにわざとセグメントは省略してあります。実際のアドレスは“セグメント：オフセット”のように表示されます。

ここでアセンブルリストを見るのが初めての人のために簡単にこのリストの読み方を説明しておきましょう。アセンブリ言語はほとんどの場合、横1行で完結します。この1行は次の順番で記述されています。

アドレス　コード　ニーモニック(アセンブリ言語)

の順です。ここで示すリストはアセンブラでアセンブル(コンパイル)した後の状態をリストとして出力したもので、その結果が16進数コードとなっています。したがって、アセンブルする前はプログラミングしたニーモニックと呼ばれるソースコードの部分だけしかありません。ある特定のメモリの内容をまったく加工せずに直接書き出すことをダンプするといいます。ここで示すのはダンプリストです。

■ダンプリスト

```
0100 EB1C          JMP 011E

011E 33C0          XOR AX,AX
0120 8ED8          MOV DS,AX
0122 8EC0          MOV ES,AX
0124 8ED0          MOV SS,AX
0126 BC8A02        MOV SP,028A
0129 FC            CLD
012A BE0B00        MOV SI,000B
012D 2EAD          LODSW CS:
012F 3D8000        CMP AX,0080
0132 7538          JNZ 016C
0134 BEB901        MOV SI,01B9

0137 B8000A        MOV AX,0A00
013A CD18          INT 18
013C B40C          MOV AH,0C
013E CD18          INT 18
0140 B412          MOV AH,12
0142 CD18          INT 18

0144 0E            PUSH CS
0145 1F            POP DS
0146 33C0          XOR AX,AX
0148 8EC0          MOV ES,AX
014A B800A0        MOV AX,A000

014D 26F606010508  TEST Byte Ptr ES:[0501],08
0153 7403          JZ  0158
0155 B800E0        MOV AX,E000

0158 8EC0          MOV ES,AX
015A BF4001        MOV DI,0140
015D AC            LODSB
015E 0AC0          OR  AL,AL
0160 7404          JZ  0166
0162 AA            STOSB
0163 47            INC DI
0164 EBF7          JMP 015D
0166 B006          MOV AL,06
0168 E637          OUT 37,AL                   ;'7'
016A EBFE          JMP 016A

0100   EB 1C 90 4E 45 43 20 32-2E 30 30 00 04 01 01 00   k..NEC 2.00.....
0110   02 C0 00 D0 04 FE 02 00-08 00 02 00 00 00 33 C0   .@.P.~........3@
```

```
02B0  0A E4 75 02 8B DE 0B F6-C3 54 68 69 73 20 69 73  .du..\^.vCThis is
02C0  20 74 68 65 20 64 61 74-61 20 64 69 73 6B 00 4E   the data disk.N
02D0  6F 20 73 79 73 74 65 6D-20 66 69 6C 65 73 00 49  o system files.I
```

これはIPL(Initial Program Loader、イニシャル・プログラム・ローダ) と呼ばれるフロッピーディスク上のトラック0セクタ0に書かれているプログラムの一部分です。この部分を読み込んでパソコンが立ち上がっていたのです。MS-DOSの解説書などを読むと予約領域などと書かれている場所の内容がこれなのです。

では、ダンプリストのアドレス0100から011Eまでをみてください。どこかでみたことのあるようなメッセージが記録されていると思いませんか? ここにはそれぞれのフロッピーディスクに関する管理情報が記録されています。ここのデータを読みとることによって、このディスクが2HDのディスクであることがわかるようになっています。

次に、逆アセンブルリストのアドレス0134を見てください。これは、015Dからのストリングコピーのために SIレジスタに、プログラムの終了点をセットしています。これはダンプリストを見ると分かりますがストリングデータの始まりでもあります。しかし、実際にこのディスクからブート (起動) してみると本来は “This is the data disk” と表示されるはずなのですが、“No system files” と表示されます。この現象はブート時の特殊な環境で発生のため追跡してみることが困難でした。

ぼくのIPLもやはりこのような現象が起こったため、試行錯誤で正しく表示される位置を探しだしました。(MOV SI,48H ; offset STRING DATA) アドレス014Dでは、資料がないので断定はできませんが、ハイレゾリューションモードかどうかのチェックをおこなっているのではないかと思います。

アドレス015Dから0166までは、ストリングデータを0かどうかチェックしながらテキストV-RAMに転送しています。これは、ストリングデータがASCIZ文字列で記録されているので、0がストリングデータの終了コードになります。

アドレス0166から016Aまではベルを鳴らしています。そしてアドレス016Aは無限ループになっています。だから音を出し続けて何もキー入力を受け付けないのです。

■リスト 11-7

```
 1: BOOT    SEGMENT AT OFFFFH
 2: ASSUME  CS:BOOT
 3:
 4:         ORG     OH
 5:
 6: RESET   LABEL   FAR
 7:
 8: BOOT    ENDS
 9:
10:
11: CODE    SEGMENT
12:
13: ASSUME  CS:CODE
14:
15:         MOV     AX,0A00H
16:         INT     18H
17:         MOV     AH,0CH
18:         INT     18H
19:         MOV     AH,12H
20:         INT     18H
21:
22:         MOV     CX,892
23:         MOV     SI,OFFSET CODEEND; 32H          ; offset STRING DATA
24:         MOV     AX,0A000H       ; CRT SEGMENT ADDRESS
25:         MOV     ES,AX
26:         PUSH    CS
27:         POP     DS
28:         XOR     DI,DI           ; CLEAR DI
29:         CLD
30:
31: LOOP1   LABEL   NEAR
32:         MOVSB
33:
34:         INC     DI
35:         LOOP    LOOP1
36:
37:         XOR     AX,AX                   ; READ KEY
38:         INT     18H
39:
40:         JMP     FAR PTR  RESET          ; SOFT RESET
41:
42: CODEEND LABEL   NEAR
43:
44: CODE    ENDS
45:
46:         END
```

以上が解析結果です。今回の改良IPLプログラムはこれらを考慮してより使いやすくしました。リスト11-7はアセンブラのソースリストです。このプログラムでは、ベルを鳴らすのをやめました。そのかわり、もっと役にたつようにメッセージを表示した後、キー入力を待って何かキーが押されたらCPUが電源ON時に実行するアドレス(リセットキーを押されたときもこのアドレスです)にFARジャンプします。簡単にいえばソフトウェアでリセットスイッチを押すのと同じことをしています。

また、PC-9801シリーズではBIOSに1文字表示ルーチンがないため、しかたなくテキストV-RAMへの直接転送をしています。このため、現状ではメッセージに使えるのはASCII文字だけです。テキストV-RAMは1バイト文字(ASCII文字)の場合は、V-RAMの偶数アドレスにだけ書き込むことで表示できますが、2バイト文字の場合は特別な処理をしなければなりません。そこで、今回は1バイト文字だけを使用することにしてIPLプログラムが複雑化するのを避けることにしました。自信のある方は2バイト文字、つまり漢字を使用できるようにしたり、さらに1バイト文字と2バイト文字との混在を可能にしてみてください。

また、デバッグには十分時間をかけるようにしてください。このプログラムは、フロッピーディスクを書き換えるためデバッグが不十分な場合、フロッピーディスクの内容まで壊してしまう可能性があります。フォーマットしたばかりでまだ何も書き込んでないデータディスクを1枚用意してやるとよいでしょう。[1]

それでは、このIPLプログラムをフォーマット済みのディスクに書き込むためのPascalプログラムをリスト11-8に示します。少し長いので間違えないように注意してください。この中には上記のIPLプログラムをバイナリーコードにして配列におさめてありますから、IPLプログラムを自分で改造しようとするのでなければ上記のアセンブラのプログラムを入力する必要はありません。

[1]安全のためハードディスクはシステムから切り離しておいてください。

■リスト 11-8

```
 1: program BootMessage;
 2:
 3: Uses Crt;
 4:
 5: type
 6:    Str80    = string[80];
 7:    DiskType = record
 8:                 SystemID          : array[1..8] of char;
 9:                 SectorSize        : integer;
10:                 CLusterSize       : byte;
11:                 ReservedCLusters  : integer;
12:                 FATs              : byte;
13:                 RootEntries       : integer;
14:                 TotaLSectors      : integer;
15:                 FormatID          : byte;
16:                 SectorsPerFAT     : integer;
17:                 SectorsPerTrack   : integer;
18:                 sides             : integer;
19:                 HiddenSectors     : integer;
20:               end;
21:
22: const
23:    Version = '1.0';
24:
25: var
26:    Fname        : str80;
27:    DiskData     : DiskType;
28:    MessageFiLe  : fiLe of char;
29:    SectorBuffer : array[1..1024] of byte;
30:
31:
32: function Exists(FiLeName : str80) : booLean;
33:
34: var
35:   f : fiLe;
36:   ResuLt : booLean;
37:
38: begin
39:   Assign(f,FiLeName);
40:   {$I-}
41:   Reset(f);
42:   {$I+}
43:   ResuLt := (IOResuLt = 0);
44:   exists := ResuLt;
45:   if ResuLt = true then
46:      CLose(f);
47: end; { of exists }
```

```
 48:
 49:
 50: procedure TiTLe;
 51:
 52: begin
 53:   cLrscr;
 54:   Write('Boot Message UtiLity');
 55:   WriteLn(' - Version ',Version);
 56: end; { of TiTLe }
 57:
 58:
 59: procedure GiveHeLp;
 60:
 61: begin
 62:    TiTLe;
 63:    WriteLn('このユーティリティは指定されたファイル中のメッセージをIPLプログラム');
 64:    WriteLn('と共にブートセクタに書き込むものです。(2HDディスク専用)');
 65:    WriteLn;
 66:    WriteLn('例  BOOTMESS TEXT.FIL');
 67:    WriteLn;
 68:    WriteLn('メッセージファイル中にはコントロールコードを含むことはできません。');
 69:    WriteLn('またVRAMに直接文字を転送するためCR、LF等やシフトJISコードも使用');
 70:    WriteLn('できません。メッセージは最大892文字まで書くことができます。');
 71:    WriteLn;
 72: end; { of GiveHeLp }
 73:
 74:
 75: procedure GetFiLeName;
 76:
 77: var
 78:   i : integer;
 79:
 80: begin
 81:   if paramcount > 0 then{ コマンドラインオプションからの読み込み }
 82:     Fname := paramstr(1)
 83:   eLse
 84:   begin
 85:     GiveHeLp;
 86:     GoToXY(1,22);
 87:     Write('メッセージファイル名 : ');
 88:     ReadLn(Fname);
 89:   end;
 90:
 91:   for i := 1 to Length(Fname) do
 92:       Fname[i] := upcase(Fname[i]);
 93: end; { of GetFiLeName }
 94:
```

```
 95:
 96: procedure GiveErrorHeLp;
 97:
 98: const
 99:   beLL = 7;
100:
101: begin
102:   if Length(Fname) > 0 then
103:     WriteLn(chr(beLL),'*** Sorry, fiLe ',Fname,' not found');
104:
105:   WriteLn;
106: end; { of GiveErrorHeLp }
107:
108:
109: procedure ReadDiskData;
110: { ディスクの識別情報をセット }
111:
112: begin
113:   with DiskData do
114:   begin
115:     SystemID         := 'TURBO---';
116:     SectorSize       := 1024;
117:     CLusterSize      := 1;
118:     ReservedCLusters := 1;
119:     FATs             := 2;
120:     RootEntries      := 192;
121:     TotaLSectors     := 1232;
122:     FormatID         := $FE;
123:     SectorsPerFAT    := 2;
124:     SectorsPerTrack  := 8;
125:     Sides            := 2;
126:     HiddenSectors    := 0;
127:   end; { of with }
128: end; { of ReadDiskData }
129:
130:
131: procedure PrepareSectorBuffer;
132: const
133: { 新しくブートセクターに書き込むＩＰＬ }
134:   PreambLeSize    = 3;
135:   MachineCodeSize = $2A;
136:
137:   PreambLe      : array[1..PreambLeSize]    of byte = ($EB, $1C, $90);
138:   MachineCode   : array[1..MachineCodeSize] of byte
139:   = (
140:       $B8,$00,$0A,{       MOV   AX,0A00H ; CRT MODE SET   }
141:         $CD,$18,  {       INT   18H                        }
142:       $B4,$0C,    {       MOV   AH,0CH   ; TEXT DISPLAY   }
143:       $CD,$18,    {       INT   18H                        }
```

```
144:        $B4,$12,      {          MOV   AH,12H    ; CURSOR OFF       }
145:            $CD,$18,  {          INT   18H                          }
146:
147:        $B9,$7C,$03,{            MOV   CX,892    ; MAX MESSEGE SIZE }
148:        $BE,$48,$00,{            MOV   SI,48H    ; offset STRING DATA }
149:        $B8,$00,$A0,{            MOV   AX,0A000H; CRT SEGMENT ADDRESS}
150:            $8E,$C0,  {          MOV   ES,AX                        }
151:        $0E,          {          PUSH  CS                           }
152:        $1F,          {          POP   DS                           }
153:        $33,$FF,      {          XOR   DI,DI                        }
154:        $FC,          {          CLD                                }
155:                          LOOP1  LABEL NEAR                         }
156:        $A4,          {          MOVSB                              }
157:        $47,          {          INC   DI                           }
158:        $E2,$FC,      {          LOOP  LOO                          }
159:
160:            $33,$C0,  {          XOR   AX,AX     ; Read KEY         }
161:        $CD,$18,      {          INT   18H                          }
162:
163:        $EA,$00,$00,$FF,$FF
164:                      {          JMP   FAR PTR $0FFFF:0000;SOFT Reset}
165:        );
166:
167: var
168:   i,
169:   CharCntr,
170:   Posn,
171:   MessageTraiLerSize : integer;
172:
173:
174: procedure ReadMessageFiLe;
175:
176: const
177:   MaxMessageLength = 892;
178:
179: var
180:   ch : char;
181:
182: begin { of ReadMessageFiLe }
183:   TiTLe;
184:   GoToXY(1,8);
185:   Write('Reading Text FiLe...');
186:   Assign(MessageFiLe,Fname);
187:   Reset(MessageFiLe);
188:   charCntr := 0;
189:   whiLe (CharCntr < MaxMessageLength) and (not EofF(MessageFiLe)) do
190:   begin
191:     Read(MessageFiLe,ch);
192:     if ch in [' '..'~'] then
```

```
193:     begin
194:       Inc(Posn);
195:       Inc(charCntr);
196:       SectorBuffer[Posn] := ord(ch);
197:     end; { of if }
198:   end; { of whiLe }
199:   CLose(MessageFiLe);
200: end; { of ReadMessageFiLe }
201:
202:
203: begin { of PrepareSectorBuffer }
204:   Move(PreambLe[1], SectorBuffer[1], PreambLeSize);
205:   Posn := PreambLeSize;
206:
207:   with DiskData do
208:   begin { prepare DiskData }
209:     for i := 1 to sizEofF(SystemID) do
210:     begin
211:       Inc(Posn);
212:       SectorBuffer[Posn] := ord(SystemID[i]);
213:     end; { of for }
214:
215:     Inc(Posn);
216:     SectorBuffer[Posn] := Lo(SectorSize);
217:     Inc(Posn);
218:     SectorBuffer[Posn] := Hi(SectorSize);
219:
220:     Inc(Posn);
221:     SectorBuffer[Posn] := CLusterSize;
222:
223:     Inc(Posn);
224:     SectorBuffer[Posn] := Lo(ReservedCLusters);
225:     Inc(Posn);
226:     SectorBuffer[Posn] := Hi(ReservedCLusters);
227:
228:     Inc(Posn);
229:     SectorBuffer[Posn] := FATs;
230:
231:     Inc(Posn);
232:     SectorBuffer[Posn] := Lo(RootEntries);
233:     Inc(Posn);
234:     SectorBuffer[Posn] := Hi(RootEntries);
235:
236:     Inc(Posn);
237:     SectorBuffer[Posn] := Lo(TotaLSectors);
238:     Inc(Posn);
239:     SectorBuffer[Posn] := Hi(TotaLSectors);
240:
241:     Inc(Posn);
```

```
242:     SectorBuffer[Posn] := FormatID;
243:
244:     Inc(Posn);
245:     SectorBuffer[Posn] := Lo(SectorsPerFAT);
246:     Inc(Posn);
247:     SectorBuffer[Posn] := Hi(SectorsPerFAT);
248:
249:     Inc(Posn);
250:     SectorBuffer[Posn] := Lo(SectorsPerTrack);
251:     Inc(Posn);
252:     SectorBuffer[Posn] := Hi(SectorsPerTrack);
253:
254:     Inc(Posn);
255:     SectorBuffer[Posn] := Lo(Sides);
256:     Inc(Posn);
257:     SectorBuffer[Posn] := Hi(Sides);
258:
259:     Inc(Posn);
260:     SectorBuffer[Posn] := Lo(HiddenSectors);
261:     Inc(Posn);
262:     SectorBuffer[Posn] := Hi(HiddenSectors);
263:
264:   end; { of with }
265:   Inc(Posn);
266:
267:   Move(MachineCode[1], SectorBuffer[Posn], MachineCodeSize);
268:
269:   Posn := Posn + MachineCodeSize -1;
270:
271:   ReadMessageFiLe;
272:
273:   MessageTraiLerSize := DiskData.SectorSize - PreambLeSize - sizEofF(DiskData)
274:                   - MachineCodeSize - CharCntr;
275:   Inc(Posn);
276:   fiLLchar(SectorBuffer[Posn], MessageTraiLerSize, 0);
277: end; { of PrepareSectorBuffer }
278:
279:
280: procedure WriteSectorBuffer;
281: var
282:   ch : char;
283:
284: begin
285:   TiTLe;
286:   GoToXY(1, 8);
287:   WriteLn('フォーマット済みのデータディスクをドライブAにセットして下さい。');
288:   WriteLn(fname,'からブートセクターにメッセージをコピーします。');
289:   GoToXY(1,22);
290:   Write('Press any key to contine...');
```

```
291:   ch := ReadKey;
292:
293:   TiTLe;
294:   GoToXY(1, 8);
295:   Write('Writeing the boot sector...');
296:
297:   inLine( $50/                 { PUSH     AX                   ; SAVE REGS   }
298:          $53/                  { PUSH     BX                                 }
299:          $51/                  { PUSH     CX                                 }
300:          $52/                  { PUSH     DX                                 }
301:          $55/                  { PUSH     BP                                 }
302:          $56/                  { PUSH     SI                                 }
303:          $57/                  { PUSH     DI                                 }
304:          $1E/                  { PUSH     DS                                 }
305:          $06/                  { PUSH  ES                                    }
306:          $FB/                  { STI                           ; EnabLe ints }
307:          $B8/>$0000/           { MOV      AX,0000              ; Drive A:    }
308:          $8D/$1E/SectorBuffer/{ LEA      BX,SectorBuffer; Source addr }
309:          $B9/>$0001/           { MOV      CX,0001              ; 1 Sector    }
310:          $BA/>$0000/           { MOV      DX,0000              ; Log Sector 0}
311:          $CD/$26/              { INT      26H                  ; Do it!      }
312:          $9D/                  { POPF                          ; Toss fLags  }
313:          $07/                  { POP      ES                   ; Restore regs}
314:          $1F/                  { POP      DS                                 }
315:          $5F/                  { POP      DI                                 }
316:          $5E/                  { POP      SI                                 }
317:          $5D/                  { POP      BP                                 }
318:          $5A/                  { POP      DX                                 }
319:          $59/                  { POP      CX                                 }
320:          $5B/                  { POP      BX                                 }
321:          $58);                 { POP      AX                                 }
322:
323:   WriteLn(^G,'　終了！');
324:   WriteLn; WriteLn;
325: end; { of WriteSectorBuffer }
326:
327:
328: begin
329:   GetFiLeName;
330:   if Exists(Fname) then
331:   begin
332:     ReadDiskData;
333:     PrepareSectorBuffer;
334:     WriteSectorBuffer;
335:   end
336:   eLse GiveErrorHeLp;
337: end.
```

5 ~ 29 行目ではグローバル変数 (プログラム全体で有効な変数) や型などの定義を行なっているだけです。間違えないように入力すれば特に気にするような部分ではありません。関数や手続きを多用していますが、それぞれの関数や手続きの働きを確実に理解していけばそんなに難しいものではないはずです。

32 ~ 47 行目までは指定されたファイルが存在しているかをチェックする関数 Exists です。この関数はよく使うので、あなたのツール集にユニットとしてつけ加えることをおすすめしておきます。Exists は指定されたファイルが存在しているときには TRUE を存在していないときは FALSE を返します。50 ~ 56 行目の手続きはプログラムのタイトルを表示するものです。

59 ~ 72 行目までの手続き GiveHelp は、使い方を説明するための手続きです。75 行目の手続き GetFileName は、このプログラムが起動されたときにパラメータとしてファイル名が指定されたかどうかをチェックし、指定されていない場合には使い方を説明する表示をします。このチェックは 81 行目で行なっています。ファイル名が指定されていた場合には、91 行目でファイル名としてセーブし、指定されていない場合には 85 行目で使い方を説明する手続きを実行した後、88 行目でファイル名の入力を促します。91、92 行目でファイル名を大文字に変換します。

96 ~ 106 行目の手続き GeveErrorHelp は、指定されたファイルが存在していないときに実行されます。ここではビープ音を鳴らして、見つからなかったファイル名を表示するだけです。

109 ~ 128 行目の手続き ReadDiskData はフロッピーディスクに書き込むための情報をセットしています。このプログラムでは変更できるフロッピーディスクが 2HD に限定されていますが、この手続きで目的のフロッピーディスクから情報を読みとるように変更すると、対象のフロッピーディスクを限定しなくてもよくなります。しかし、今回はこれ以上プログラムを複雑にしたくなかったので、よく使われている 2HD ディスクに限定しました。このデータはフロッピーディスクのトラック 0 セクタ 0 に記録されているので、他のディスクを使いたい人はデバッガなどでこのデータを取り込んで、手続きを変更してください。

131 行目が実際の IPL プログラムにジャンプさせるためのコードです。140 ~ 163 行目までが新しい IPL プログラムを配列にしたものです。コメントの形で、アセンブリ言語での記述を残してありますが、MASM でアセンブルできる形のリストは既に紹介してあります。174 ~ 200 行目の手続き ReadMessageFile では、指定されたメッセージファイルの内容を読み込んで、記録する IPL プログラムを格納してある配列に追加しています。ここで注意が必要なのは 177 行目の定数です。この定数はメッセージの長さを指定しています。この長さはセクタの長さが 1024 バイトあるうちの IPL プログラムのサイズやフロッピーディスクの管理情報などが使用しますから、これらのサイズを差し引いたサイズです。安全のためにすこしサバよんでいるので、計算してもピッタリ

一致しません。

203 ~ 277 行目がいよいよこの手続きの本体です。ここでは、フロッピーディスクに書き出すイメージを作成しています。280 ~ 325 行目の手続き WriteSectorBuffer は目的のフロッピーディスクの決まった位置に、作成されているイメージを書き出すためのものです。307 ~ 311 行目までがフロッピーディスクへの直接書き込みを行なっているところです。ここでは INT26H を使ってこの機能を実現しています。なぜ INTR 手続きを使わずに INLINE 文を使ったかというと、INT25H と 26H は割り込みをかける前と後とではスタックのレベルが違うのです。レベルが違うというのは、スタックポインタの位置が異なるということです。このようなことが起こる理由は割り込み処理が終了するときに、元のフラッグがスタックに積まれているためです。これをそのままにしておくと暴走してしまうので、POPF などを実行してスタックのレベルを合わせておく必要があります。このような細かな記述はアセンブリ言語でないと記述できないため、このように INLINE 文を使って実現せざるとえないのです。

このプログラムのように Inline を使わずに、アセンブラで書いた手続きや関数を TURBO Pascal のプログラムやユニットとリンクするには、OBJ(ファイル) にしておかねばなりません。そのための方法や注意は TURBO Pascal のプログラマーズ・ガイドに実例を含め詳しく説明がありますから、そちらを参考にしてください。

328 行目からがこのプログラムのメインルーチンで、ここではメッセージファイルがあるかないかで処理が分かれているだけです。

少し難しかったと思いますが、理解できたでしょうか。理解できなかったひともくじけずに何度か読み返してみてください。どうしてもわからなければ、しばらくしてまた時間がたってから読んでみると、ああそうかとわかることがあります。自分の技術が知らない間に上がっているので理解できた、ということがぼくの経験でもよくありました。最後に、ぼくが作ったメッセージファイルを以下に示します。

```
Not a system disk or MS_DOS disk!
Replace the disk and press any key when ready
```

今まで、このメッセージファイルについての注意には触れていませんでしたが、このファイルの中にはコントロールコードを含むことができません。この原因は PC-9801 の CRT BIOS(バイオス) に 1 文字表示ルーチンがないためです。ブートしている途中では、BIOS しか使うことができません。このために、MS-DOS のファンクションコールも使えないのです。そこでやむをえず直接テキストを V-RAM に表示するコードを書き込んでいるのですが、この方法ではキャリッジリターン (CR)、ラインフィード (LF) などのコントロールコードがそのまま表示されてしまいます。したがって、このメッセージを入力するときには、行の区切りにリターンキーを押してはいけません。CRT は横

が80文字ですから、80文字に満たない場合にはスペースを入力していって80文字にしてください。892文字までメッセージとして登録できるので、このように記述しても十分間に合うはずです。簡単にいうと、ここで使えるメッセージは1行だけということです。

ここで重ねていっておきますが、このようなプログラムになってしまったのはぼくのせいではなく、必ずしもできのよくないPC-9801のせいといったらいい過ぎでしょうか。工夫をすればもっと使いやすいプログラムにはなりますが、プログラムが複雑になってしまうのでサンプルプログラムのようになったというわけです。このままでもピー音が鳴らなくなるので助かるはずです。

このプログラムを作っていて感じたことは、データディスクになぜどうしようもないような「不親切プログラム」を書き込んでいるのだろうかという疑問です。たったこれだけ(ここで紹介しているプログラムのこと)でかなり便利になるのにと思われてしかたありません。

MS-DOSでこの領域を操作しているのは、ぼくが知っている限りではFORMATコマンドしかありません。つまり、FORMATコマンドであの「不親切プログラム」を書き込んでいるわけです。そこで、FORMATコマンドを改造することも考えられますが、それにはかなりの技術とフロッピーディスクコントローラ(FDC)のノウハウが必要なのでこれ以上はやりません。

最後に大切な注意をしておきます。

このプログラムで加工を対象としているのはAドライブです。したがって、あなたのパソコンのハードディスクがAドライブである場合はハードディスクが破壊されるので厳重な注意が必要です。かならずフロッピードライブをAドライブとなるように立ち上げてからこのプログラムを動かしてください。うっかり統合環境からいきなりこのプログラムを走らせるとハードディスクが壊されますから、くれぐれも注意してください。

12 効率アップのためのテクニック

最後の仕上げに

これまでさまざまなサンプルプログラムを使って、いろいろなトレーニングを重ねてきましたが、いよいよこれが最後の章です。ここでは少し目先を変えて、開発のテクニックのノウハウを最後のまとめということにします。

長い時間をかけて苦労したあげくに完成したプログラムであっても、実行速度が遅すぎて使い物にならなくて、結局プログラム全部をC言語やアセンブリ言語で書き直す羽目になるということは結構あります。

この章では、実行に時間がかかりすぎる部分だけを高速化し、すべてを書き直さなくても済むような効率のよいTURBO Pascalプログラムを開発するためのテクニックを紹介します。

バージョンアップも怖くない

12-1■上位バージョン(ver6.0)への移植

TURBO Pascalの人気が爆発的に高まったのはver3.0でした。ver3.0はかなり長い間使われていたことなどから、ver3.0で書かれたプログラムをたくさん持っている人も多いはずです。Borland International 社でもバージョンアップ用のツールをTURBO Pascalに標準添付しています。

これから当分の間はver3.0から4.0への変更のようなドラスティックなことはないようですし、なんといってもバージョンアップにより実行速度が向上していますから、手持ちの古いver3.0のプログラムをver6.0に対応させておきたいものです。ここではそのためのツールの使用法を解説します。

ではまず、以下のver3.0用のプログラムをver6.0用に移植してみましょう。

■ Ver3Prog.pas

```
 1: program Ver3Prog;
 2:
 3: var
 4:   startsec,
 5:   endsec    :integer;
 6:
```

```
 7: procedure Timer(var second:integer);
 8: type
 9:   Register=record
10:            ax,bx,cx,dx,bp,si,di,ds,es,frags:integer;
11:            end;
12: var
13:   reg : Register;
14:
15: begin
16:   reg.ax := \$2c00;
17:   Msdos(reg);
18:   second := Trunc(reg.cx/256)$*$3600+(reg.cx mod 256)$*$60
19:     +Trunc(reg.dx/256);
20: end;\{of Timer\}
21:
22: begin
23:   CLrScr;
24:   Timer(startsec);
25:   DeLay(10000);
26:   Timer(endsec);
27:   WriteLn( 'Time: ',endsec-startsec);
28: end.\{of Main\}
```

ジャーナルファイルを作成して、移植のためのくわしい情報を得ることにします。

```
upgrade /j VER3PROG
```

と入力します。すると次のような画面が表れます。これはファイルの拡張子 (.PAS) を 3TP のように変更するための許可を求めています。ここでは、y とタイプします。

```
UPGRADE Version 6.0 Copyright (c) 1990,91 Borland International
Warning:  all input files will be renamed.  Enter Y to continue
```

すると、続いて以下のようなメッセージが表示され、upgrade は終了します。

```
Scanning for upgrades...
VER3PROG.PAS(28)
Inserting USES statement...
VER3PROG.PAS(39)

1 warning reported
```

ここでディレクトリをとってみましょう。ちゃんとジャーナルファイル (.JNL) と拡張子 3TP のファイルができています。

```
ドライブ D: のディスクのボリュームラベルはありません.
ディレクトリは D:¥LIST

VER3PROG 3TP 494 87-02-02 15:14
VER3PROG JNL 577 90-01-29 0:51
VER3PROG PAS 893 90-01-29 0:51
3 個のファイルがあります.
808960 バイトが使用可能です.
```

そこでジャーナルファイルを見てみます。

```
1.  VER3PROG.PAS(17)
Msdos(reg);
      ^
Parameter to MsDos must be of the type Registers defined in DOS unit.
MsDos の引数は Dos ユニットで定義されている Registers 型でなければなりません.

Turbo3.0 とは異なり, Turbo6.0 では, MsDos 手続きに渡す引数に対して厳格
な型チェックが行なわれます. MsDos の引数は, Dos ユニットで定義されている
Registers 型でなければなりません.

UPGRADE は自動的に, 引数を正しい型に型キャストします.

詳しくはリファレンスガイド第 12 章を参照してください.
```

ソースファイルは以下のように変更されていました。

■ソースファイル

```
 1: {$R-}    {Range checking off}
 2: {$B+}    {Boolean complete evaluation on}
 3: {$S+}    {Stack checking on}
 4: {$I+}    {I/O checking on}
 5: {$N-}    {No numeric coprocessor}
 6: {$M 65500,16384,655360} {Turbo 3 default stack and heap}
 7:
 8: program Ver3Prog;
 9:
10: Uses
11:   Crt, {Unit found in TURBO.TPL}
12:   Dos; {Unit found in TURBO.TPL}
13:
14: var
15:   startsec,
```

```
16:   endsec    :integer;
17:
18: procedure Timer(var second:integer);
19: type
20:   Register=record
21:        ax,bx,cx,dx,bp,si,di,ds,es,frags:integer;
22:        end;
23: var
24:   reg : Register;
25:
26: begin
27:   reg.ax := $2c00;
28:   Msdos(Dos.Registers(reg));
29: {! 1. Pa^rameter to MsDos must be of the type Registers defined in DOS unit.}
30:   second := Trunc(reg.cx/256)$*$3600+(reg.cx mod 256)$*$60
31:     +Trunc(reg.dx/256);
32: end;{of Timer}
33:
34: begin
35:   CLrScr;
36:   Timer(startsec);
37:   DeLay(10000);
38:   Timer(endsec);
39:   WriteLn( 'Time: ',endsec-startsec);
40: end.{of Main}
```

このソースファイルをジャーナルファイルの内容を参考にして ver6.0 用に移植したのが VER6PROG.PAS です。よく見くらべてみてください。

■ Ver6Prog.pas

```
 1: {$R-}     {Range checking off}
 2: {$B+}     {Boolean complete evaluation on}
 3: {$S+}     {Stack checking on}
 4: {$I+}     {I/O checking on}
 5: {$N-}     {No numeric coprocessor}
 6: {$M 65500,16384,655360} {Turbo 3 default stack and heap}
 7:
 8: program VER6PROG;
 9:
10: Uses
11:   Crt, {Unit found in TURBO.TPL}
12:   Dos; {Unit found in TURBO.TPL}
13:
14: var
15:   startsec,
16:   endsec   : word;        { integer; }
17:
```

```
18: procedure Timer(var second : word);       { integer; }
19: var
20:   hour, minute, sec, sec100 : word;       { GetTime の引数には word 型を使う }
21:
22: begin
23:   GetTime(hour, minute, sec, sec100);     { PC-9801 では sec100 は必ず 0 }
24:   second := hour$*$3600 +minute$*$60 +sec;
25: end;{of Timer}
26:
27: begin
28:   CLrScr;
29:   Timer(startsec);
30:   DeLay(10000);
31:   Timer(endsec);
32:   WriteLn( 'Time: ',endsec -startsec);
33: end.{of Main}
```

ver3.0 にはなかった手続きや関数がかなり増えています。特筆すべきことは、MS-DOS のファンクションコールを使わなければ実現できなかったようなことが、標準の(TURBO Pascal では標準という意味。標準 Pascal にはこのようなものはない) 手続きや関数として用意されたということです。

移植せずに ver6.0 でプログラムをコンパイルしたいというひとは upgrade の起動時に/3 のスイッチをつけることをお奨めします。こうすると ver3.0 と高い互換性を持つユニット Turbo3 が付加されます。しかし、このユニットもいつまで標準添付されるかという心配もありますし、せっかくバージョンアップで増えた機能を使えないのは惜しいことです。後々のことを考えれば ver6.0 で走るように移植しておきたいものです。

ver3.0 から ver6.0 への最もやっかいな変更はなんといっても、標準手続き Read 、Readln の動きが変わってしまったことです。この変更を吸収するためにも/3 のスイッチは有効でしょう。しかし、このスイッチは最後の手段としてとっておくようにしたほうが、よりはやく新しい Read 、Readln になれることができるでしょう。

開発管理のためのノウハウ

12–2■プログラムもプロジェクト管理

だんだん腕が上がって、複雑で大きなプログラムを作るようになると あれやこれやをすべて自分で管理しなければならないのでそのために長い時間が必要になります。特にハードディスクの中にあれこれすべての関数や手続きを入れておくと目的の関数、手続きを探すのが大変になったり、IDE(統合開発環境) でのコンパイルやデバッグがメモリ不足で不能になったりします。

また、数人で分担してプログラミングする場合では、当然ソースファイルを分割して、いくつかのファイルを使って開発しなければならなくなります。このため、作成したソースファイルが複数になり、どのプログラムにどのソースファイルが必要なのかという情報を管理していく方法が必要になります。

この管理方法には、ユニットごとに整理する方法、IDE に組み込まれている MAKE、BUILD といった機能を使う方法、さらに、MAKE コマンド (ユーティリティ) を使う方法が用意されています。ちょっと特殊な方法としてはソースファイル内の記述を条件によって切り変える「条件コンパイル」ということも可能です。

12-2-1■ユニットごとに整理する方法

TURBO Pascal ではユニットがサポートされています。これにより、プログラムの分割コンパイルが可能になっています。このことは、プログラムをパーツ化して作成し、残しておけるという大きなメリットがあります。用途別に整理してユニット名をつけるようにすれば、プリンタを使うときにはこのユニット (たとえば PRINT)、サウンドを使うときにはこのユニット (たとえば SOUND) などと容易に使い分けることができるようになります。ユニットの記述方法については第 4 章を参照してください。

さて、1 本のプログラムだったものをユニットに変更する手順を簡単に説明します。以下のようなプログラムを例にして考えてみましょう。

```
PROGRAM  TEST;

PROCEDURE A;
BEGIN
 :
END;

PROCEDURE B;
BEGIN
 :
END;

FUNCTION C (ZZZ : WORD) : WORD;
BEGIN
 :
END;

BEGIN
 :
END.
```

このプログラムの手続き B と関数 C をユニットにまとめることにしましょう。まず、このソースファイル (TEST.PAS) を作成したいユニット名 (TESTUNIT.PAS) でコピーします。これ以降、編集するファイルは TESTUNIT.PAS です。次に、手続き B と関数 C 以外の部分はエディタなどを使って削除します。すると以下の部分だけになります。

```
PROCEDURE B;
BEGIN
  :
END;

FUNCTION C (ZZZ : WORD) : WORD;
BEGIN
  :
END;
```

この状態のファイルに、TURBO Pascal にユニットであることを認識させるための書式をファイルの先頭に数行追加します。

```
UNIT    TESTUNIT;

 INTERFACE

 PROCEDURE B;
 FUNCTION C (ZZZ : WORD) : WORD;

 IMPLEMENTATION
```

このうち、PROCEDURE B と FUNCTION C (ZZZ : WORD) : WORD; の部分にはこのユニットから呼び出して使用したい手続きや関数を記述します。ここでは手続き B と関数 C を他のプログラムから呼び出して使えるユニットにしたいので、このような記述になります。最後に、このファイルの一番後に end. を追加します。もしも、ユニットを使用する前に初期化したい変数などがあるときは、この end. の前に begin を追加して、この begin と end. のペアの間に初期化のプログラムを記述します。

これでユニット化は終了です。コンパイルしてエラーがなければ*.TPU というファイルがディスクに作成されているはずです (この例では TESTUNIT.TPU)。

ユニットを作成するうえでの注意することがあります。それは、ユニットにはコンパイル後のマシンコードが 64KB 以内でなければならないという制限です。

12-2-2■IDE の MAKE、BUILD を使う方法

IDE は統合開発環境をうたうだけあって様々な便利な機能を備えています。この中にはプロジェクト管理を行うことができる MAKE や BUILD も含まれています。

MAKE とはユニットのソースファイル (*.PAS) とオブジェクトファイル (*.TPU) の日付と時間をチェックしてソースファイルのほうがオブジェクトファイルよりも新しい場合 (ソースファイルに変更があり、コンパイルを忘れているような場合)、再コンパイルを自動的に行ってくれるという機能です。

ユニットはプログラムを作成していくうちに進化して行きますから、ソースファイルには関数を追加しても、コンパイルするのを忘れてしまうこともあります。しかし、コ

ンパイルを忘れると、実際に使用するオブジェクトにはこの関数が含まれていないのでエラーになってしまいます。

TURBO Pascal の MAKE 機能では次の 3 種類のチェックを行っています。

(1) メインプログラムで使われているユニットのソースファイルとオブジェクトファイルの日付と時間をチェックして、ソースファイルが修正されている場合は再コンパイルする。

(2) 変更したユニットのインタフェース部が修正されたかをチェックする。

もしも修正されていた場合、そのユニットを使用している他のユニットを再コンパイルします。実現部だけを修正したときには再コンパイルは行われません。

(3) ユニットが使用しているインクルードファイルやコンパイラ指令 ($L) でリンクしている.OBJ ファイルが修正されている場合、再コンパイルする。

このようなチェックはとても面倒な気がしますが、IDE では f・9 キーでこのような煩わしいチェックを自動的に行ってくれます。この機能がなかったらすべてのチェックを自分で行わなくてはならないのかと思うとぞっとします。TPC.EXE でコンパイルしたときには、MAKE 機能を含めるために/M オプションをつけて起動します。

BUILD は MAKE の特殊な場合といえます。この機能を使うと対象のプログラムで使用しているすべてのユニットが再コンパイルされます。意識的にすべてのユニットを最新のものにしたいときに有効な機能です。IDE でこの機能を使うには Compile メニューで Build を選択します。TPC.EXE では/B オプションをつけて起動します。

12-2-3■MAKE コマンド (ユーティリティ)

TURBO Pascal では IDE に MAKE 機能をもっていますが、コマンドラインコンパイラ TPC.EXE を使用したい人やアセンブリ言語で作成した.OBJ ファイルなどをリンクしている場合、最新のオブジェクトファイルかどうかをチェックしたいときには IDE の MAKE 機能は役に立ちません。

このようなときには別に用意されている MAKE コマンドを使用します。同じ名称のコマンドが MS-DOS を購入すると付属してきますが、ここで紹介するのは Borland International 社の MAKE コマンドです。この 2 つは基本的には同じことを行うためのコマンドなのですが、作成者 (社?) が異なるためまったく同じに使うというわけにはいきません。ここでは Microsoft 社の MAKE コマンドについては説明しませんが、興味のある人は両者の違いを調べてみるとおもしろいかもしれません。Borland International 社は TURBO シリーズで Microsoft 社の Quick シリーズに対抗しスピードを売りものにしているので MAKE のスピードを比較してみてもおもしろいでしょう。

IDE ではユニット名と同じ名前の*.PAS ファイルをソースファイル、同じ名前の*.TPU ファイルをオブジェクト (ユニット) ファイルとしていたので、特にソースファイルとオブジェクトファイルとの関係を IDE に知らせる必要はありませんでしたが、アセン

ブルしたオブジェクトファイルをリンクしている場合を想定すると、MAKEコマンドにソースファイルとオブジェクトファイルの関係を知らせる必要があります。

このような目的のテキストファイルをメイクファイルと呼び、以下のような書式で記述します。このファイルはMAKEFILEという名前で作成しておくとMAKEコマンドが自動的に読み込んで実行してくれます。

```
testar.exe: teststar.pas testdefs.pas testlib.pas ¥
              test.obj
      tpc teststar /m

test.obj: test.asm
      tasm test.asm test.obj
```

まずTESTSTAR.EXEがTESTSTAR.PAS、TESTDEFS.PAS、TESTLIB.PASとTEST.OBJによって構成されていることを表しています。1行目の“¥”は改行を無視して、次の行に記述が続いていることを示します。長い記述をだらだらと続けると後で読むときに長い時間を必要としますが、このマークを使って行を区切ると、ソースプログラムの種類(Pascal、C、アセンブリ言語など)別に分けるといった意味付けが簡単に行えます。このとき、簡単なコメントをつけておきたければ、まず、行を変えて先頭に“#”(半角文字で)と空白をつけます。この後にコメントを続けて記述すればよいのです。

さて、話を元に戻します。3行目はTESTSTAR.EXEを作成するための方法、つまり、コンパイルやアセンブルの方法が記述されます。この例ではコマンドラインコンパイラTPC.EXEのMAKE機能を使うために/Mを指定しているのでTESTDEFS.PASやTESTLIB.PASについてのコンパイル方法についての記述が必要ありません。

次のTEST.OBJについての記述はこのオブジェクトファイルがTEST.ASMによって構成されていることを表しています。その次の記述はもう見当がつくでしょう。そうです。この行はTEST.OBJを作成するための方法(アセンブルのしかた)について記述されています。TASMというのはBorland International社製のアセンブラです。もっと詳しい使い方はTURBO Pascalに添付されているユーザーズガイドにあります。高度な使い方にチャレンジしたい人はそちらを一度読むことをおすすめします。

12-2-4■条件コンパイル

1種類のソースプログラムで様々な状況に対応するプログラムを作成しようとするとするのは非常に難しいことです。たとえば、PC-9801用のプログラム、IBM-PC用のプログラム、FM-R用のプログラムを基本的な処理が同じだからというような理由から1つのプログラムとして管理しようとすると、この条件コンパイル機能を使わないで実

現すると大変なことになってきます。各機種ごとにプログラムファイルとメイクファイルを用意しなければならなくなるからです。

そこで条件コンパイル機能が登場します。条件の指定方法はTURBO Pascalで使用しているコンパイラ指令によく似ているので違和感なく使うことができるでしょう。次に使用することができる条件を示します。

{$DEFINE　シンボル}	他の条件指令で使うシンボルを定義
{$UNDEF　シンボル}	定義済みシンボルを除去
{$IFDEF　シンボル}	シンボルが定義済みなら続く部分をコンパイル
{$IFNDEF　シンボル}	シンボルが未定義なら続く部分をコンパイル
{$IFOPT　x＋}	xがONなら続く部分をコンパイル
{$IFOPT　x－}	xがOFFなら続く部分をコンパイル
{$ELSE}	直前の$IF**が真なら続く部分をコンパイル
{$ENDIF}	$IF**や$ELSEの終了を示す

以下、順にくわしくみていきましょう。

(a) $DEFINE、$UNDEF指令

この指令は指定されたシンボルが定義されているかどうかによって処理を変える$IFDEFや$IFNDEFといった指令を有効に機能させるためにシンボルを定義したり、除去したりする指令です。

$DEFINE指令ではシンボルを定義し、$UNDEF指令ではシンボルを除去(未定義の状態に)します。また、シンボルはコンパイルするときに定義することも可能です。

シンボルを定義するには$DEFINEを使って、

{$DEFINE　シンボル}

という1行をソースファイル内に記述します。シンボルは通常使用する識別名と同じように長さ、許される文字などに制限があるので注意してください。たとえば、

{$DEFINE　DEBUGS}

とソースファイル内に記述すると、シンボル“DEBUGS”が定義されます。この定義はこれ以降の部分で有効になります。しかし、$UNDEF指令でこのシンボルが除去された場合、$UNDEF指令以降は無効になります。

このシンボル“DEBUGS”を除去する$UNDEF指令は以下のように記述します。

{$UNDEF　DEBUGS}

さらに、コマンドラインコンパイラでコンパイルを行うときは/Dオプションを指定することで同様のシンボルを定義することができます。以下にTEST.PASをコンパイルするときにシンボルDEBUGSとIBMを定義する方法を示します。

```
TPC   TEST   /DDEBUGS   /DIBM
```

IDE(統合環境) でシンボルを定義するには、O/C/Conditional Defines オプションを使います。シンボルをいくつか定義したい場合にはシンボルとシンボルの区切りにセミコロンを使用します。上記の例と同じシンボルを定義するには、入力ボックスが表示されたら次のように入力します。

```
DEBUGS；IBM
```

シンボルには TURBO Pascal が自動的に定義しているものがいくつかあり、ユーザーのプログラミングを助ける働きをしています。このシンボルはユーザーが定義したものと同じようにチェックすることができます。

VER**	いつも定義されている (**の部分はバージョン 6.0 では 60 となる)
MSDOS	いつも定義されている
CPU86	いつも定義されている
CPU87	コンパイル時に 8087 が存在しているときに定義される

VER**(たとえば VER60) シンボルはコンパイラのバージョンの違いを吸収します。つまり、将来機能が拡張されたときでも異なるバージョン用のプログラムを 1 本のソースファイルで管理することができます。

MSDOS シンボルは MS-DOS 上でコンパイルが行われているときに定義されます。つまり、現状では TURBO Pascal は MS-DOS でしか動作しませんからいつも定義されていることになります。

CPU86 シンボルはインテルの 8088 、8086 、80186 、80286 、80386 、80486 といった Intel iAPX86 シリーズの CPU を使用したコンピュータでコンパイルが行われているときに定義されます。つまり、このシンボルも現状ではいつも定義されていることになります。

CPU87 シンボルはデータ型に SINGLE 、DOUBLE 、EXTENDED 、COMP といった IEEE 浮動小数点型を使用するときに、8087 数値演算コプロセッサが存在しているかどうかをチェックするときに使うことができます。

このシンボルはコンパイル時に設定されるものなので、あなたの友だちなどに実行ファイル (*.EXE) を作ってあげるような場合には使わないほうがよいでしょう。

(b) $IFDEF 、$IFNDEF 、$IFOPT 、$ELSE 、$ENDIF 指令

この指令はパスカルの IF 文と同じように働きます。つまり、

```
{$IF*** シンボル}
TEXT1
{$ELSE}
TEXT2
{$ENDIF}
```

というような記述があるとき、{$IF*** シンボル }の条件が真のときはTEXT1がコンパイルされ、TEXT2はコメントと同じ扱いになります。その反対に条件が偽のときはTEXT1がコメントの扱いになり、TEXT2がコンパイルされます。

つまり、{$ELSE}はパスカルのELSEと同じ役割を果たします。{$ENDIF}はBEGINやENDにあたるものがないため、TEXT2の終了をコンパイラに知らせる役割をしています。パスカルでのENDのことだと思っていればよいでしょう。また、この構文は16レベルまでネストすることができます。

$IFDEF指令と$IFNDEF指令は反対の意味で使われます。つまり、上記のような例を$IFDEFと$IFNDEFで記述したとすると、$IFDEFを使ったときはシンボルが定義されているときTEXT1がコンパイルされ、$IFNDEFを使ったときはシンボルが定義されていないときTEXT1がコンパイルされます。

$IFOPT指令はコンパイラオプションをチェックするときに使用します。たとえば、Nオプションをチェックして8087があるときにはDOUBLE型を使用して計算精度を上げ、ないときにはREAL型を使用して計算スピードを上げたいようなときには次のように変数を定義することも可能です。

```
VAR

{$IFOPT   N +}
   TEST    :   DOUBLE;
{$ELSE}
   TEST    :   REAL;
{$ENDIF}
```

$IFOPT指令は{$IFOPT　N-}という記述も認めていますから、この記述法を使うと上記の例は逆になり、

```
VAR

{$IFOPT   N-}
   TEST    :   REAL;
{$ELSE}
   TEST    :   DOUBLE;
{$ENDIF}
```

となります。プログラムするときに分かりやすいほうを選んで使い分けるとよいでしょう。

$IFOPT指令でチェックできるコンパイラオプションは以下の通りです。

```
$A, $B, $D, $E, $F, $G, $I, $L, $N, $O, $R, $S, $V, $X
```

12-2-5■ユーティリティの使い方

TURBO Pascal にはプログラム開発に必要、または便利なユーティリティが付属してきます。ここでは、TPUMOVER、MAKE、TOUCH、GREP、BINOBJ といったあまり聞いたことがないようなものについて説明していきます。

(1) TPUMOVER

このユーティリティは主に TURBO.TPL というコンパイル時に使用するシステムユニットを編集するためのものです。ユニットファイルには.TPU と.TPL という 2 種類の形態が用意されています。

ユーザーが作成するユニットファイルはコンパイルすると*.TPU というファイル (TURBO Pascal Unit) を作成しますが、このファイルは 1 つのユニットでできています。しかし、*.TPL というファイル (TURBO Pascal Library) は複数のユニットを持つことができます。つまり、TURBO Pascal コンパイラが使用する TURBO.TPL は、一般的に必要なユニット (標準ユニット) をすべて含んだユニットのライブラリなのです。

このファイルはコンパイル時に必ず使用するのでメモリ内にロードされてしまいます。そのため、コンパイルの対象となるファイルが大きい場合にはコンパイルするのに必要なメモリが足りなくなるといったことが起こります。そのようなとき、普段使用しないユニットを TURBO.TPL から取り除くことで使用可能メモリを確保することができます。このファイルの中に含まれているユニットは SYSTEM、PRINTER、CRT、DOS、OVERLAY の 5 つです。

さて、TURBO.TPL はこのくらいにして、次は編集するためのプログラム側について説明します。このプログラム (TPUMOVER.EXE) はメニュー形式で編集対象のユニットを選択した後、追加するのか、削除するのかを指定するだけで OK、という非常に使い勝手のよいプログラムに仕上がっているため、使い方についての心配はまったく必要ありません。使っていてわからないことがあっても、f･1 キーを押すことで使用するキーの説明や簡単な例が表示されるという統合環境並みの親切さです。

```
TPUMOVER ファイル名1 ファイル名2
```

と入力すると、画面上に左右に 1 つづつ、2 つのウィンドウが表示され、ウィンドウの一番上にはファイル名が表示され、ウィンドウの中にはそのファイルが持っているユニットの一覧表が表示されます。どちらのウィンドウが選択されているかは、ユニットを選択するためのバーが表示されているかですぐにわかると思います。また、PC-98 版ではウィンドウの枠が少し太くなります (IBM-PC 版では 2 重線になる)。

また、ファイル名 1 は左側、ファイル名 2 は右側のウィンドウに表示されます。この 2 つのファイル名は指定しなくてもよく、ファイル名 2 が省略されたときは右側のウィ

ンドウには "NONAME.TPU" という名前がつけられます。2つとも省略されたときは左側にはTURBO.TPLを読み込み、右側のウィンドウには "NONAME.TPU" という名前がつきます。また、指定されたファイル名が見つからないときは*.TPLというファイル名の一覧表を表示します。

このユーティリティで使用できる主なキーと機能は画面の一番下に表示されますが、それらは次の9つです。

f・1	ヘルプ画面の表示
f・2	選択されているウィンドウのファイルをセーブする
f・3	選択されているウィンドウに新しいファイルを読み込む
f・4	バーのついているユニットの詳しい情報を表示する
f・6	ウィンドウの選択
+(プラス)	ユニットのマークと解除
INS	マークされているユニットのコピー
DEL	マークされているユニットの消去
ESC	TPUMOVERの終了

f・1、f・2、f・3、f・6、ESCキー以外のものについてもう少し説明しましょう。

f・4キーを押すと、ポップアップウィンドウと呼ばれる別のウィンドウが開き、この中にバーがついていたユニットの名前、コードサイズ、データサイズ、シンボルの大きさ、依存するユニットなどの情報が表示されます。実はこれらの情報は普段から表示されているのですが、ウィンドウの大きさの問題で依存するユニットの表示が1つに限定されているのです。この場所に表示されているユニット名の後にピリオドが3つ続いているときは依存するユニットが複数存在していることを示します。したがって、このキーを使うのは依存するユニットをすべて知りたいときだけということになります。

+キーはコピーや消去の対象となるユニットを指定するのに使います。対象となるユニットは複数選択することが可能です。マークの方法は始めに+キーを押すとマークされ、もう一度押すとマークが解除されるといった簡単なものです。

INSキーを押すと、マーク済みのユニットが選択されていないほうのウィンドウにコピーされます。ここが間違えやすいところなのですが、マルチウィンドウに対応しているエディタなどではウィンドウを選択しなおして、コピーしたいウィンドウを選択してからコピーを行いますが、このユーティリティでは選択されていないウィンドウにしかコピーできないというちょっと変わったつくりになっているので注意が必要です。

DELキーを押すと、マーク済みのユニットが選択されているウィンドウから削除されます。さらに、TURBO.TPLを編集するには画面上での操作だけでなくコマンドラインから直接編集することも可能で、その書式は以下の通りです。

```
TPUMOVER　TURBO　/パラメータ　ユニット名
```

ユニット名には追加、削除したいユニットの名前を記述します。パラメータには+、-、*、?の4つが用意されています。

/+	TURBO.TPL に指定したユニットを追加する
/−	TURBO.TPL から指定したユニットを削除する
/*	TURBO.TPL から指定したユニットを取り出し、ユニット名.TPU というファイルにセーブする
/?	ヘルプ画面を表示する

(2) MAKE

IDE の MAKE 機能でも説明しましたが、MAKE ユーティリティは MAKE ファイルと呼ばれる指示ファイルが必要となります。つまり、いくら MAKE ユーティリティを持っていても、この MAKE ファイルを作成する技術がなければ使えません。

TURBO Pascal のディスクの ¥TP¥UNITS の中に入っている MSA 社から提供されている RS232C.MAK を取り出してみてください。これを例にしましょう。

拡張子が MAK になっているファイルをメイクファイルといいます。このファイルはエディタでテキストファイルとして作ります。それではさっそく RS232C.MAK の内容を見てみましょう。先頭の 2 行はそれぞれ TASM 、MASM という名称のマクロの内容を定義しています。たとえば、1 行目ではマクロ TASM は TASM /ml と同じものとなります。3 行目はマクロ ASM がマクロ TASM の展開であることを定義します。

5 行目では MAKE の対象になるファイルを示します。7 行目で RS232C.OBJ は RS232C.ASM に依存しており、その依存関係は 8 行目に書いてあるように、ターボアセンブラに ml オプションをつけてアセンブルしたものであることを示します。$* は拡張子を除くことを示します。10 行目はユニット RS232C.TPU がどのファイルに依存するのかと、コマンドラインコンパイラ TPC でどのようにコンパイルすればよいのかを示しています。13 行目もまったく同様です。

■ RS232C.MAK

```
 1: TASM    = TASM /ml
 2: MASM    = MASM -ml
 3: ASM = \$(TASM)
 4:
 5: ALL: RS232C.OBJ RS232C.TPU RSDEMO.EXE
 6:
 7: RS232C.OBJ: RS232C.ASM
 8:     \$(ASM) \$$*$;
 9:
10: RS232C.TPU: RS232C.PAS RS232C.OBJ
11:     TPC /M RS232C.PAS
12:
13: RSDEMO.EXE: RS232C.TPU RSDEMO.PAS
14:     TPC RSDEMO.PAS
```

結局、メイクファイルは TURBO Pascal でのプログラムのコンパイル手順を記述したバッチファイルなのです。手順をファイルに記述しておけば、後で再コンパイルする

必要が生じたときに、古いファイルを使ったり、どこかが抜けてエラーが出てわずらわされるといったことがなくなります。プログラムを開発したらすぐにメイクファイルを作っておくのは、複数のTPUファイルやOBJファイルを合成する場合、非常に有効です。

このメイクファイルを実際に走らせるには、MS-DOSのコマンドラインに対して、

```
MAKE -fRS232C
```

と入力します。MAKEユーティリティは非常に利用価値の高いものです。ぜひTURBO Pascalのユーザーズガイドに目を通しておくようお勧めします。なお、MAKEユーティリティ自体にヘルプ機能がありますから、MAKE -?または-hとして走らせて出力される表示を参照してみてください。

(3) TOUCH

ファイルの更新作成日時をMS-DOSが今管理している日時に変更してくれます。あるプログラムファイルを構成している他のプログラムファイルをTOUCHすると、MAKEユーティリティによりそのプログラムファイルから作られるEXEファイルの日時が更新されます。

(4) GREP

複数のファイルの中から指定されたテキストを探し出すものです。プログラムが複数のファイルを合成して作られている場合に、ある特定の手続きがどこにあるとか、ある入出力をどのファイルでするのかを探すときに非常に便利です。GREPにもヘルプ機能があります。

(5) BINOBJ

ファイルの種類にかかわらずどんなファイルでもTURBO Pascalに手続きとしてリンク可能な.OBJファイルに変換してくれます。このユーティリティのありがたみ(?)を実験するためにTURBO Pascalの中にサンプルプログラムBGILINK.PASが例として提供されています。この例では決まったバイナリファイルを読み込まなければならないプログラム部分をBINOBJにより外部手続きに変更し、バイナリデータごとexternal宣言される手続きにする方法を示しています。

ユーティリティは、ここで説明したことだけですぐに使えるというわけには行きませんが、これほどの機能を使わない手はありません。それにTURBO Pascalを購入したときの価格の中にこのユーティリティも含まれているのです。お金にきびしいパソコンマニアとしてはTURBO Pascalを100%使いこなしたいものです。

12–2–6■TURBO Cとのリンク

マニュアルにはアセンブリ言語とのリンクの記述はありますが、C言語とのリンクに関する記述はありません。しかし、アセンブリ言語とのリンクに使用した$L指令はインテル・リロケータブルオブジェクトモジュールフォーマット(OBJ)の形式を持ったファイルとTURBO Pascalとをつなぐためのものでした。したがって、OBJ形式のファイルをはきだすコンパイラならば、どのようなコンパイラでも可能であろうと考えてチャレンジした結果、以下の方法を使ってリンクすることに成功しました。

まず、初めに用意するものが2つあります。1つはTURBO Cコンパイラ、もう1つはTURBO Pascalのディスク中にあるTURBO.CFGかCTOPAS.TCです。あなたが統合環境版を使っているのならCTOPAS.TCを、コマンドライン版を使っているならばTURBO.CFGを用意してください。このファイルは、Cコンパイラのオプションを決定するためのものです。

次のリストを見てください。これは、XのY乗を求めるプログラムです。

■X-Y

```
 1: /* program XのY乗 */
 2:
 3: typedef unsigned int     word;
 4:
 5: word mult(word x, word y)         /* XのY乗 */
 6: {
 7:    word  count,
 8:          result= 1;
 9:
10:    for(count=1; count <= y; count++)
11:        result *= x;        /* result := result * x */
12:
13:    return(result);
14: }
15:
16: program LinkC;  { TURBO Cとのリンク }
17:
18: {$L mult.obj}           { リンクするobjファイル }
19:
20: function mult(x,y : word) : word;     external;
21:
22: begin
23:   writeln('2の4乗= ',mult(2,4));
24: end.
```

始めのリストがC言語のもので、次のリストがPascalのものです。このプログラムはC言語で記述した関数をPascal側から呼び出すだけのものです。ここでは、2の4

乗を求めています。C 言語の 1 行目では、変数の型を定義しています。Pascal の TYPE 宣言と同じです。C 言語には WORD という型がないため、同じ意味の unsigned int 型を WORD と宣言しています。そのあとは、mult という WORD 型の関数を定義して、その内容を記述しているだけのものです。

Pascal では$L 指令を使って Pascal とリンクする OBJ ファイルを指定して、このファイル中に記述してある関数の型や引数などを宣言しています。ここで重要なのは external という予約語です。これは、サブプログラムが他のモジュール (この場合 mult.obj) に記述してあることを TURBO Pascal コンパイラに伝えるためのものです。

さて、このプログラムのコンパイル手順ですが、それは以下のようになります。

(1) TURBO C で C 言語のプログラムをコンパイルする。

統合環境版の場合は、次のように入力します。

```
TC /CCTOPAS.TC MULT.C
```

コマンドライン版では、次のように入力します。

```
TCC MULT.C
```

(2) TURBO Pascal で Pascal プログラムをコンパイルする。

このようにしてみると、とても簡単に見えます。

しかし、注意することが 2 つあります。1 つは、C プログラム中には、C のランタイムライブラリを使用する関数は使えません。通常これらの関数は、実行ファイルを生成する際にリンカが各ライブラリからリンクしてくるために、OBJ ファイル中にはこれらの関数のコードが含まれないからです。絶対にこれらの関数が使えないのかというと、ちゃんとこれを回避する方法が用意されています。これらの関数が使えないのはライブラリからリンクしてくるためなので、なんとかしてソースコードとして書いてしまえばコンパイルしたときに OBJ ファイルにオブジェクトコードとして書き込まれます。しかし、これらの関数を自分で作っていたのでは膨大な時間がかかりますし、バグも当然発生します。このような問題をクリアするためには Borland International から発売している C 言語のランタイムライブラリのソースコードを入手すればよいのです。当然ながら、このソースコードは現在使っているライブラリのものであるため、できあがったモジュールは (入力ミスなどのバグがあればそれも含めて) まったく同じ動作をします。

もう 1 つの問題は、TURBO C のバージョンです。ver2.0 の場合は TURBO Pascal に用意されている CTOPAS.TC、TURBO.CFG が使えるのですが、ver1.5 の場合、私

が試した限りではTURBO Pascalに付属のものは使えませんでした。そのかわりに、TURBO C(ver1.5)についていたものは使うことができました。

さて、ここまでを理解できた人ならば、手続き(Procedure)をTURBO Cで記述することも簡単にできるはずです。TURBO Pascalディスクの中にあるCPASDEMO.C、CPASDEMO.PASの中にも説明があります。

通常はこのような使い方ですむはずです。ところが、ユニットの中でスピードを向上させた処理にしなくてはならない場合もあると思います。同様にユニット内に記述すればよいと考えらえますが、実はこの方法ではうまくいきません。

少し意地になって調べてみると、ユニットを使うやり方と今までのやり方ではどうもメモリモデルが異なるようです。今までの方法はメインプログラムのなかにリンクしていますが、この状態でTURBO Pascalが使えるメモリはリファレンスガイドのTURBO Pascalの内部構造に書いてある通りデータ、コードの各セグメントは64KBまでです(ユニット等を使用するとき、コード全体のサイズは64KBを越えられます)。このために今までの方法ではTURBO Cでコンパイルする際、スモールモデルを用いました。

しかし、ユニット内にリンクしようとすると話が違ってきます。ユニット内部でのメモリの使われ方はマニュアルに記載されていないため推測の域を出ませんが、1つだけはっきりしていることは8086系のCPUを使っていることによると思われます。

TURBO Pascalでは、プログラム中、ユニットの実現部で宣言された手続き、関数はNEARコールとしてコードに展開され、ユニットのインタフェース部で宣言された手続き、関数はFARコールになります。このため、TURBO Cでコンパイルする際にメモリモデルを変更する必要があります。実験的に調べた結果、NEARコールしているところではコンパクトモデルを、FARコールしているところではラージモデルを使用するとうまくいきました。

これは、ある程度予想できました。FARコールしているところでは当然、コードもデータも別セグメントにあると思われるのでラージモデル(両方ともFARポインタ)を使用すればよいだろうと予想しました。これは当たったのですが、NEARコールしているところでは、みごとに外れてしまいました。NEARコールしているところは、コードもデータも64KBの制限内なので、今までの方法と同じスモールモデルを使用すればよいと考えたのです。ところが、暴走してしまったのです。

次に考えたのは、ミディアムモデルです。これは、ユニットを使うことによってコードが64Kの制限を越えるからではないかと考えたためです。ところがというか、やはり、というべきかこのメモリモデルでもうまく行きませんでした。

順番から行くと次はコンパクトモデルです。これはもう半信半疑でした。なぜかといえば、データが64Kの制限を越える理由が考えつかなかったためです。ところが、うまくいってしまったのです。これには、ずいぶんと悩まされました。先ほどのTURBO Pascalの内部構造のところを読み返してみてください。プログラムが実行している間

は、DS(データセグメント) レジスタの内容は変化しないので、データのサイズは64K を越えられないことになっているのです。ここで、矛盾が生じてしまったのです。

DSは不変のはずだが、ミディアムモデルで暴走するということは、ユニット内ではDSを変更しているのではないか? などと考えました。結局のところ、結論らしい結論は得られませんでした。しかし、リンクができたのでよしとしました。時間とツール(TURBO DEBUGGER 等)があるかたは、ぜひ調べてみてください。

以上で、TURBO Cとのリンクの解説を終わりますが、特に難しいことはなかったでしょう。普通にリンクするのであればスモールモデルを、ユニット内だけで使う関数、手続きをリンクするときはコンパクトモデルを、他のユニットからも使いたい、関数、手続きをリンクするときにはラージモデルを用いてPascal型の呼び出しをするようにオプションを設定すればよいのです。このテクニックを使うことでぼくは1時間30分もかかっていたあるプログラムを30分以内に終わらせることに成功しました。じつに3倍ものスピードアップを簡単に実現することができたのです。

虫干しテクニック

12–3■デバッガ

高速化テクニックは参考になりましたか。これからのあなたにはきっと役に立つでしょう。しかし、高速化する前にプログラムが完全(どのように完成されているように見えるプログラムにもバグは潜んでいます)に動作していなければいけません。さもないと、高速化した意味がありません。せっかく高速化した部分にバグがあったのでは修正がまた必要になります。そこで、デバッグ、つまりバグをとるための効率的な方法が必要になります。ここでは、デバッグの方法について考えてみましょう。

コンピュータサイエンスでも結論の出ていない方法論(デバッグのことです)についての考察を行おうというのですから、この方法はぼくや仲間達の経験的なものにすぎないということを始めにお断りしておきます。まず、現在のデバッガについて考えてみましょう。現在、入手しやすいデバッガには以下のようなものがあります。

Microsoft 社

DEBUG(デバッガ)

SYMDEB(シムデブ: シンボリックデバッガ)

CV(コードビュー)

Borland International 社

TURBO DEBUGGER

これ以外にも多くのソフトハウスから独自の機能を持ったデバッガが数多く発表されていますが、ここではそれらをすべて説明するのはとても無理ですから、上記のものに

ついてのみ説明することにします。

さて、この中でも歴史的な背景や扱うプログラムの形態で分類することができます。初めて登場したのが“DEBUG”です。このころはプログラムの開発はごく限られた人(プロのプログラマやハイレベルのパソコンマニア) しか行なわなかったためもありましたが、デバッグツールも技術的にあまり進んでいなかったので、1ステップずつ実行していくというようなデバッグしかできませんでした。

この後、少し改良されたものが“SYMDEB”です。“少し”と書いたのには理由があって、この改良はメモリ上に確保した変数エリアをシンボル (変数名) で参照することができるもので当時としてはかなり画期的なものでした。

さて、最後に登場するのが最新鋭のデバッガ CV(コードビュー) とTURBO DEBUGGER です。このデバッガではコンパイラに対応していて、コンパイラが生成した実行可能なコードとその基であるソースコード (Pascal プログラムやCプログラムなどのこと) とが1対1に対応しているので、DEBUG のときのようにバグを見つけてからソースコード上のどこに対応しているのかを探す手間が省けるようになりました。それだけでなく開発時の論理ミス (ロジックミス) を見つけるのに威力を発揮します。

このタイプのデバッガは実行ファイル (*.EXE) 内にデバッグのための情報を持つので、作成中のプログラムすべてをコンパイルして実行ファイルを作成しなければなりません。このため、メインメモリから (MS-DOS システムが占める領域+デバッガの使用するサイズ) 分のサイズ以下のプログラムしかデバッグすることができません。

デバッガのみで約 200KB を占めますから、メインメモリ 640KB のシステムでは日本語入力の FEP などのデバイスドライバを含めると 300KB を下回るサイズのターゲットプログラムしかデバッグできないことになります。コンパイラ言語 (C や Pascal など) では、コンパイル後のサイズが 200KB や 300KB 程度のプログラムはすぐに生成されてしまいます。これは、コンパイラが関数などのライブラリやデバッグ情報をプログラムに付加して実行ファイルを生成するからです。

バグが発見される度にソースプログラムを修正して、何度もコンパイルしなければなりません。この点では、IDE に付属のデバッガのほうが操作しやすいといえるでしょう。しかし、IDE にはコンパイラの本体が含まれているため、デバッガ単体のときよりもデバッグできるプログラムサイズが少なくなってしまいます。このような状況を考慮にいれて最もデバッグのしやすい環境は何かと考えた結果、

「プログラムを部品として考えてデバッグすること」

が非常に大切なことを再確認しました。

デバッグのノウハウはこれだけです。でもこのことは忘れられがちなのです。皆さんはちゃんと関数や手続きごとにデバッグしていますか。リストの上だけのデバッグでは自分だけの思い込みで OK を出してしまうことがよくあります。このため、バグの発

見を遅くしてしまうのです。ですから、IDE内のデバッガで実行しながらデバッグしてみてください。きっと、思ったよりも後の(全体の)デバッグが楽にできるはずです。

では、関数や手続きごとのデバッグをどのようにしたらよいのでしょうか。それを説明しましょう。関数や手続きは目的があって作るものですから、この使用目的に合うような値をセットして呼び出すための仮のメインプログラムを作成してみるのです。このときにターゲットとなる関数や手続きのあるファイルと同じファイル内に作成してもよいのですが、TEST.PASのように別のファイルに作成することをお勧めします。

別のファイルに作成したほうがよい理由を考えてみましょう。仮のメインプログラムはあくまでも「仮」なので実際に実行させるときには削除しておく必要があります。けれども、同じファイル内にこの仮のプログラムを作成した場合、削除し忘れてしまうことがあったり、削除しなくてもよいところまで削除してしまう、などの失敗を防ぐことができるのです。また、このように作成しておくと実際の動きと同じような動作をするので、より本番(実際)に近い状態でデバッグすることができます。

このような理由から、ぼくは部品ごとにデバッグした後に全体のデバッグをします。そしてIDEのデバッガにプログラム全体が載りきらないときに初めて、TURBO DEBUGGERやCV(コードビュー)のようなデバッガを使うようにしています。かなり大きなプログラムを作らない限り、IDEのデバッガで用が済んでしまいます。

さて、デバッグの考え方は以上で理解できたと思います。そこで、いよいよTURBO PascalのIDEのデバッガのコマンドについて説明します。大きくわけてデバッガの機能は3つあります。プログラムの実行、値(変数)の監視や変更、停止位置(ブレイクポイント)の設定です。実行と停止位置には深い関係があるので2つに分けてもよかったのですが、プログラムのデバッグをする場合には実行するよりも停止させることのほうが難しいので、あえてこのように分類してみました。

さて、ではIDEとTURBO DEBUGGERにどのようなコマンドがあるのかを主なコマンドだけに限定し、以下に分類して示します。ここでは簡単に入力ができる“ホットキー”と呼ばれるコマンドだけを示すことにします。

プログラムの実行	
f・7	1ステップ毎の実行
f・8	1ステップ実行(手続き、関数を1ステップとみなして実行)
f・4	カーソル位置までの実行
CTRL-f・9	実行(TURBO DEBUGGERではf・9)
値(変数)の監視、変更	
CTRL-f・4	値の参照、設定
CTRL-f・7	Watchウィンドウへの追加
停止位置(ブレイクポイント)の設定	
CTRL-f・8	ブレイクポイントの設定、解除
その他の機能	
CTRL-f・3	コールスタック
CTRL-f・5	実行中画面イメージとの切り替え

ぼくが主に使っているコマンドはこのくらいです。つまり、わずか9つのホットキーを覚えるだけで快適にデバッグが行うことができます。

では、実際にデバッグを行なうときの手順について考えてみましょう。

■ sum-dbg

```
 1: function Sum( i : integer) : integer;
 2:
 3: var
 4:   Total , c : integer;
 5:
 6: begin{ of Sum }
 7:   for c := 1 to i do
 8:      Total := Total + c;
 9:
10:   Sum := Total;
11: end; { of Sum }
```

このリストがデバッグするターゲットプログラムです。このリストの関数Sumは1からiまでの値を加算するというものです。次に「仮」のメインプログラムを示します。

■ main-dbg

```
1: program test;
2:
3: {$i sum.pas}
4:
5: begin
6:   writeln('Total = ',sum(10));
7: end.
```

ここではターゲットプログラムをインクルードして、ターゲットプログラムを取り込んでいますが、関数や手続きをパーツとしてとらえるならば、ここはユニットにしておいたほうがよいでしょう。この「仮」のメインプログラムでは複雑なことをしないのがデバッグを簡単にするコツです。

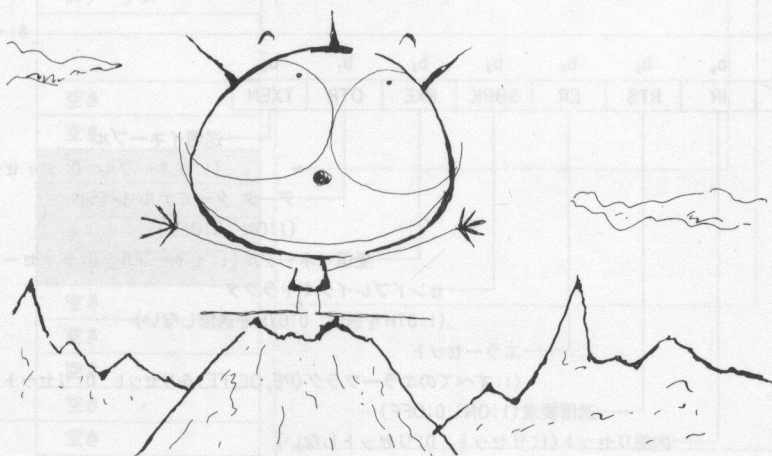
まず、「仮」のメインプログラムを「1ステップ毎の実行(f･7)」で実行します。「1ステップの実行(f･8)」で実行してしまうと、Sumが計算されてしまうためです。これ以降はコマンドを(f･*)と表します。プログラムの実行が関数Sumに移ったら、(CTRL-f･7)でWatchウィンドウに変数Totalとcを表示させてください。初めに表示されている値は、今までメモリ上にあった値ですからゴミだと思って気にしないでください。

さて、(f･7)で1ステップ毎に実行していくとFORループで止まっているように見えます。実はこれでよいのです。ループとは「ぐるぐる回る」ということなので、iの

回数だけこの加算処理が繰り返されているのです。ループの内容が2ステップ以上ある場合にはFORループ内のBEGINとENDの間を繰り返し実行することになります。

ソースコードでデバッグする利点は、このように実行している様子が目に見えることです。ここまで実行して、注意深い人は気がついたと思いますが、Totalの値がおかしいのです。実はTotalの初期化をしていないためにこのような現象が起こってしまったのです。このようなミスはよくあるので十分気をつけてください。では、このバグをとるために変数Totalに0をセットする文をFORループの前に追加してください。

関数Sumの修正が終わったら同じように(f・7)で実行してみてください。今度は正解の“55”が表示されたはずです。むろん、タイプミスなどなく正しくプログラムを入力できたとしての話です。実際に“55”が表示されているかの確認は、(CTRL-f・5)で実行中の画面と切り替えてみてください。この画面で“55”が表示されていたらOKです。「仮」のメインプログラム内の関数Sumに引き渡す値を10から他の値に変えて何回か実行してください。その間、おかしな答えを表示しなければデバッグは完了です。

この例は非常に簡単なものでしたが、デバッグの手順は複雑なものでも同じです。バグのない部分を順に確認していきます。つまり、このことは逆にバグを見つけることになるのです。焦らずに、順序よく1ステップづつ確認していってください。

AL	00H	01H	02H	03H	04H	05H	06H	07H
伝送速度(BPS)	75	150	300	600	1200	2400	4800	9600

(ALに08H以上の値を設定した場合は1200BPS)

表10-1 伝送速度

表10-2

表10-3 μPD8251Aコマンド指定

＊FLAG：RS-232C BIO用内部フラグフィールドの内容　INIT：RS-232Cインタフェース（μPD8251）の初期化済み　BFULL：受信バッファがFULL　BOVF：受信バッファオーバーフローが発生　XON：XON処理（CTRL-S出力）を行う（3/4）　XOFF：XOFF処理（CTRL-Q出力）を行う（1/4）

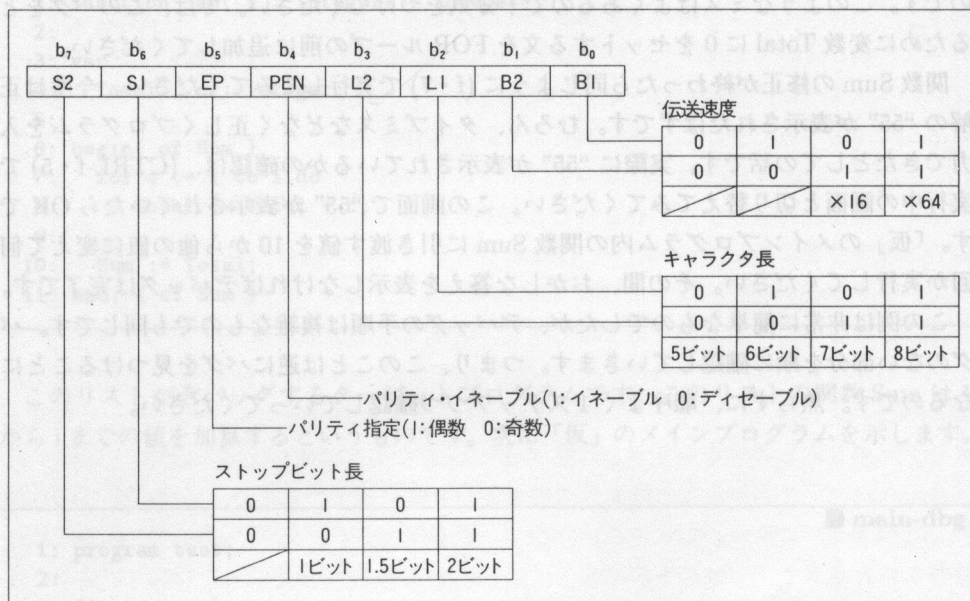
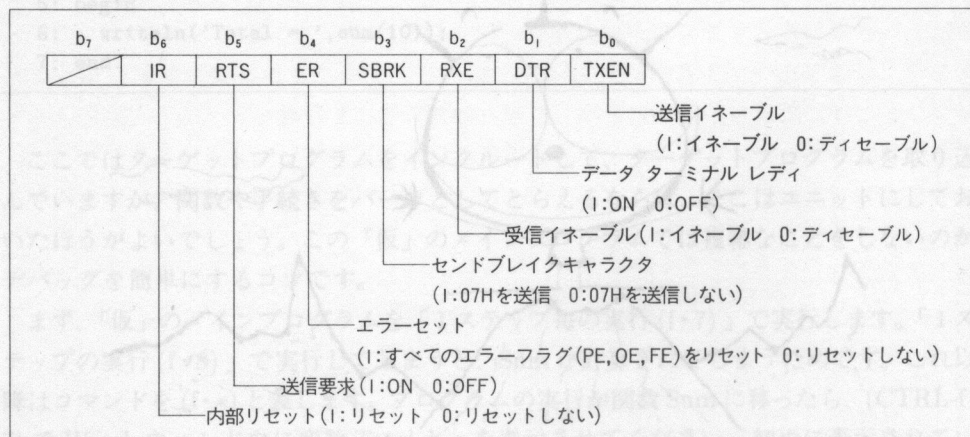
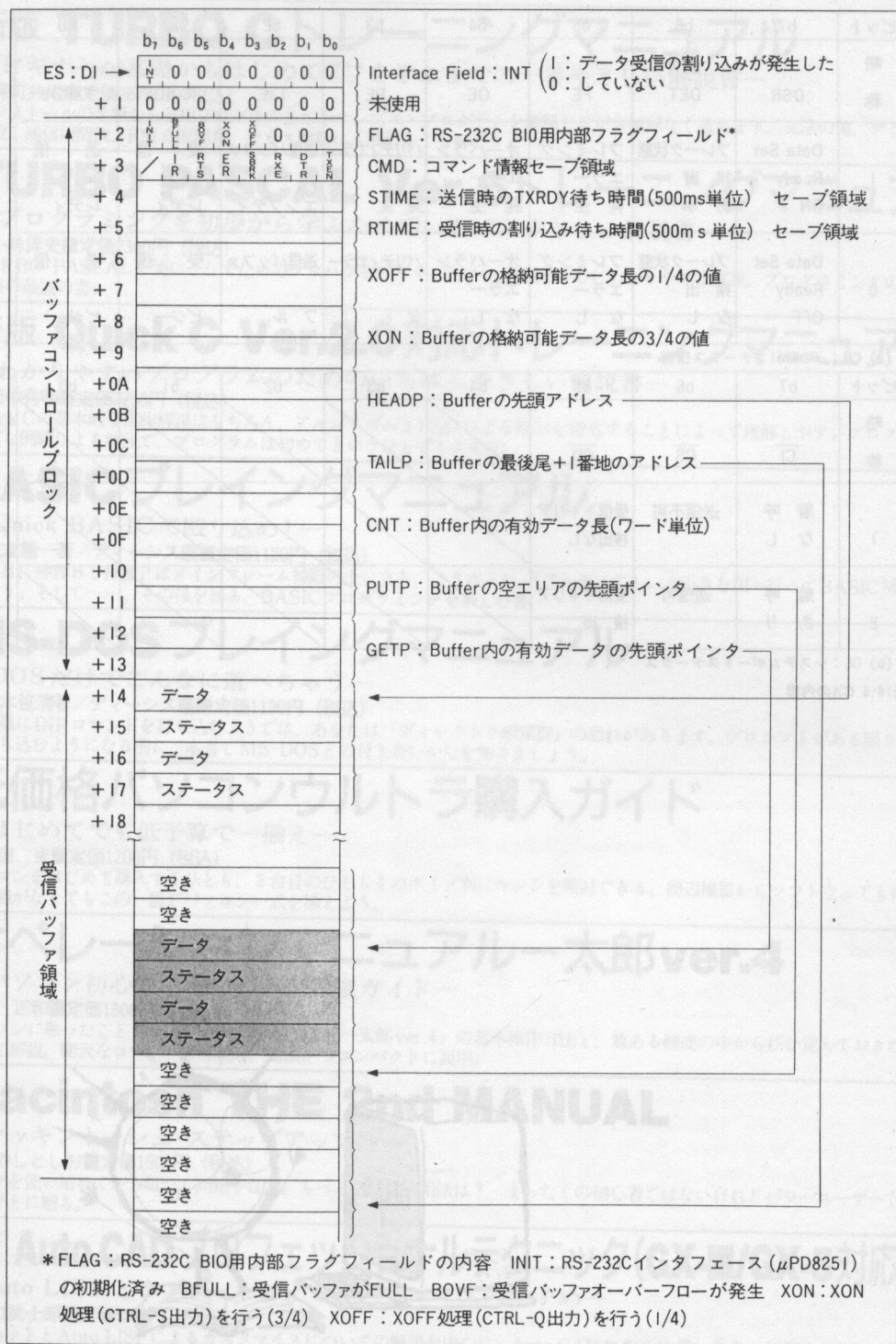



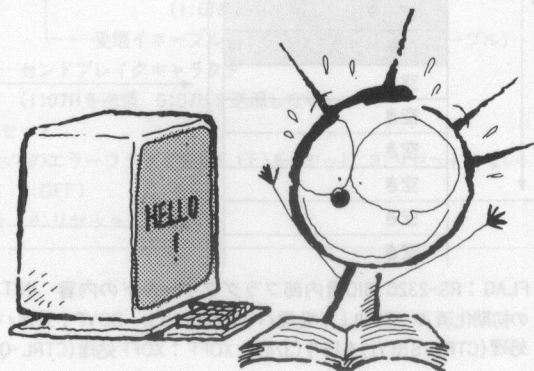
表10-4 受信バッファ形式

ビット	b7	b6	b5	b4	b3	b2	b1	b0
略称	DSR	SYN DET	FE	OE	PE	TXE	RXRDY	TXRDY
1	Data Set Ready ON	ブレーク状態検出あり	フレミングエラー発生	オーバランエラー発生	パリティエラー発生	送信バッファエンプティ	受信レディ	送信レディ
0	Data Set Ready OFF	ブレーク状態検出なし	フレミングエラーなし	オーバランエラーなし	パリティエラーなし	送信バッファフル	受信ビジー	送信ビジー

(a) CH：μPD8251ステータス情報

ビット	b7	b6	b5	b4	b3	b2	b1	b0
略称	$\overline{CI}$	$\overline{CS}$	$\overline{CD}$					
1	着呼なし	送信不可	受信キャリア検出なし					
2	着呼あり	送信可	受信キャリア検出					

(b) CL：システムポートステータス

表10-5 CXの内容

JICC出版局のコンピュータ関連書籍

新版 TURBO Cトレーニングマニュアル

―ビギナーが基礎からはじめてC++を学ぶまでの一番やさしい解説書―

■深町共栄■定価2670円（税込）

インストールの手順から応用プログラムまでをサンプル・プログラムを参照しながら無理なく進めます。文法の他、グラフィックス関数、通信用関数、FM音源関数、マウス関数にも触れることができます。

TURBO PASCAL Ver.6トレーニングマニュアル

―プログラミングを初歩から学ぶためのいちばんやさしい解説書―

■小林倖史■定価2360円（税込）

TURBO PASCALを使いこなすためのノウハウをビギナー向けに、例題を中心にわかりやすく解説。プログラミングの基礎を学ぶための最適の書。

新版 Quick C Ver.2.0対応トレーニングマニュアル

―わかりやすいプログラムのためのいちばんやさしい解説書―

■田中秀和■定価2750円（税込）

Quick Cの基本的な操作解説はもちろん、アルゴリズムとPADによる検討を徹底することによって理解しやすいプログラムを作成。豊富な例題による解説で、プログラムは初めてというひとでも大丈夫。

BASICプレイングマニュアル

―Quick BASICで殴り込め！

■大友龍一著／ディーシス編■定価1120円（税込）

ある日、神官Bと神官Pはメインフレーム神殿の上に立ち、こう言った。「アルティアという小さな国へ行ってBASIC神の教えを広めよう」そして……、その後を語る、BASICプログラミングを楽しむ書。

MS-DOSプレイングマニュアル

―DOSだけでこんなに遊べちゃう―

■島本笹清著／ディーシス編■定価1120円（税込）

無意識にDIRコマンドを打ち込むようでは、あなたは「ディレクトリ症候群」の恐れがあります。プロンプトがある限り必ずDIRと打ち込むようになる前に、本書でMS-DOSとの付き合いかたを知りましょう。

低価格パソコンウルトラ購入ガイド

―はじめてでも低予算で一揃え―

■荻窪　圭■定価1200円（税込）

パソコンをはじめて購入するひとも、2台目のひともそのタイプ別にマシンを検討できる。周辺機器からソフトウェアも掲載。予算や知識がなくてもこの一冊でパソコン一式を揃えよう。

オペレーションマニュアル一太郎ver.4

―パソコン初心者のための完全操縦ガイド―

■森　正和■定価1800円（税込）

パソコンに触ったこともないという人でもOK。「一太郎ver.4」の基本操作方法と、数ある機能の中からぴひ覚えておきたい機能を選んで解説。膨大なコマンドの解説も、簡潔かつコンパクトに説明。

Macintosh THE 2nd MANUAL

―マッキントッシュ・ステップアップ術―

■はやしとしお■定価1600円（税込）

マックを使い始めてから気づく問題や疑問。もっと便利な活用法は？　まったくの初心者ではないけれどパワーユーザーでもないというひとに贈る。

新版 Auto CADプロフェッショナルテクニック（GX-III/GX-5対応）

―Auto LISPとタブレットによるカスタマイズのすすめ―

■江口武士郎■定価3100円（税込）

タブレットとAuto LISPによるカスタマイズについての解説を中心に、Auto CADをさらに使いやすいものにするためのノウハウの紹介や著者の手によるプログラムも掲載。

《著者紹介》

小林　倖史 (こばやし・ひとし)

1944年、東京生まれ。早稲田大学理工学部電気工学科卒業。工学博士。米国クラークソン工科大学研究員を経て、現在埼玉工業大学電子工学科教授。コンピュータ科学専攻。PUG(パーソナルコンピュータ・ユーザーズ・グループ) 主宰。
著書に『電気回路理論』『産業用マイクロコンピュータの応用技術』『分散マイクロコンピュータシステム』『TURBO PASCALトレーニングマニュアル』『クイックマニュアル5：ターボパスカル』などの他、高度情報化社会とコンピュータ犯罪についての訳書、評論多数。また、『森のクラフトマン』(夫人と共著)『くたばれ、ニューヨーク』『USAキャンパス情報源』(訳) など専門外の分野でも活躍。

カバーデザイン
馬場かおる

カバーイラスト
伊藤博幸

本文イラスト
盛本康成

企画・構成
マイクロランド

■PUG BOOKS

ターボパスカルエキスパートマニュアル (ver6)

1991年8月15日　第1版第1刷発行
1992年4月6日　第2刷発行
著　者　　小林　倖史
発行人　　蓮見　清一
発行所　　JICC(ジック) 出版局
〒102 東京都千代田区麹町 5-5-5
【営業部】03-3234-4621
【編集部】03-3221-1997
郵便振替　東京 7-170829(株) ジック
印刷所　　東京書籍印刷株式会社

ISBN4-7966-0159-7